Vues et transformations de programmes pour la modularité des évolutions

Akram Ajouli

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

MINES Nantes
Institut Mines-Télécom

Thèse de Doctorat

Akram AJOULI

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'École nationale supérieure des mines de Nantes
sous le label de l'Université de Nantes Angers Le Mans

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique, section CNU 27
Unité de recherche : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Soutenue le 25 Septembre 2013
Thèse n° : ED 503-2013EMNA0112

Vues et Transformations de Programmes pour la Modularité des Évolutions

JURY

Rapporteurs :	**M. Christophe DONY**, Professeur des universités, Université Montpellier 2
	M. Jean-Louis GIAVITTO, Directeur de recherche au CNRS, IRCAM
Examinateurs :	**M. Benoit BAUDRY**, Chercheur, INRIA
	M. Jérémie Christian ATTIOGBE, Professeur des universités, Université de Nantes
Directeur de thèse :	**M. Jean-Claude ROYER**, Professeur des écoles des mines, École des mines de Nantes
Co-directeur de thèse :	**M. Julien COHEN**, Maitre de conférence, Polytechnique Nantes

Remerciements

J'exprime mes remerciements à mon encadrant, M. Julien Cohen pour les consignes qu'il m'a donné, les expériences qu'il m'a fait acquérir et pour le temps qu'il m'a consacré pour atteindre les objectifs de cette thèse. Je remercie également mon directeur de thèse, M.Jean-Claude Royer pour ses conseils et la chance qu'il m'a donné pour profiter de son expérience tout le long de cette thèse.

Je tiens à remercier également M.Christophe Dony et M.Jean-Louis Giavitto, pour avoir accepté d'être les rapporteurs de ma thèse. Je remercie aussi M.Jérémie Christian Attiogbe et M.Benoit BAUDRY pour avoir accepté d'être les examinateurs de cette thèse.

Je remercie chaleureusement les membres de ma famille qui m'ont soutenus pour atteindre ce niveau d'étude. Un grand merci pour ma mère, à mon père qui m'a encouragé à continuer mes études et bien sûr sans oublier mes frères, ma sœur, mes oncles, ma grande mère Dada et tous les autres.

Table des matières

Table des figures

1

Introduction

Sommaire

1 Problème de la tyrannie de la décomposition dominante

Dans l'industrie environ 80% du coût de développement des logiciels est dépensé dans leur évolution [Erl00]. En effet, les grands projets logiciels ont une longue durée de vie et sont donc amenés à être continuellement adaptés. Ceci est formalisé par des lois d'évolution [BL71, Leh96], notamment :

Croissance continue. Le nombre de fonctionnalités du logiciel est toujours en croissance afin de garantir la satisfaction des besoins des utilisateurs.

Continuité des changements. Un logiciel doit être sans cesse adapté à son contexte d'application (qui change continument) pour rester satisfaisant aux besoins des utilisateurs.

Diminution de la qualité perçue. L'incertitude et les imprévus du monde réel peuvent diminuer la qualité perçue des logiciels. Ceci est dû au fait qu'un utilisateur peut voir d'autres logiciels qui sont plus satisfaisants que le sien. De ce fait l'utilisateur peut considérer son logiciel comme moins satisfaisant.

La facilité d'évolution des logiciels dépend de la façon dont ce logiciel est structuré. Ainsi, un programme écrit de manière monolithique sera plus difficile à faire évoluer qu'un programme qui est décomposé en des parties relativement indépendantes appelées *modules* [Par72]. Chaque module rassemble des éléments qui sont dépendants et communique avec les autres modules à travers une interface. Ceci permet d'opérer certaines maintenances de manière modulaires, c'est à dire en se limitant à la modification d'un module donné.

La modularité facilite donc la maintenance et permet ainsi de réduire le coût de maintenance. La plupart des langages de programmation fournissent des supports à la modularité via des constructions syntaxiques et des vérifications statiques ou dynamiques, comme les méthodes/procédures/fonctions, les classes/objets en programmation objet, les bibliothèques, les paquetages, les traits [DNS+06], les mixins [BC90], les foncteurs [Ler00], les composants [Szy02] et les aspects [KH01].

La modularité d'un programme est le résultat des choix architecturaux lors de la conception. Toutefois, une autre loi d'évolution énonce la conséquence des évolutions successives sur la modularité d'un code source :

Augmentation de la complexité. La complexité des logiciels augmente si les actions successives d'évolutions ne sont pas guidées et contrôlées. Ainsi pour conserver les propriétés de modularité de l'architecture initiale, on doit implémenter les corrections tout en respectant cette architecture et non pas juste les implémenter pour faire au plus simple et au plus vite. On parle de *dégénérescence de l'architecture*.

Notons également que quelle que soit la structure d'un programme, certaines tâches de maintenances seront modulaires et d'autres ne le seront pas.

Considéreons par exemple le programme suivant.

```
abstract class Expr{
 abstract int eval();
 abstract String show();
}
```

```
class Num extends Expr{
 Integer x;
 int eval(){return x;}
 String show(){return x.toString();}
}
```

```
class Add extends Expr{
 Expr e1, e2;

 int eval(){return e1. eval() + e2.eval();}
 String show(){return e1.show() + e2.show();}
}
```

Ce programme permet de représenter et manipuler des expressions arithmétiques. Voyons cette hiérarchie de classes comme un type de données Expr constitué de deux possibilités : Num pour des constantes et Add pour des additions. Deux fonctionnalités sont possibles : le calcul de la valeur d'une expression, et l'affichage d'une expression.

L'entité modulaire utilisée ici est la classe. On trouve une classe *abstraite* qui définit le type de données, et une classe pour chaque constituant du type de données. Chaque classe implémente les deux méthodes eval et show. Le code qui définit le comportement des fonctions (ici des méthodes) sur les constantes est défini dans la classe Num et le code définissant le comportement des fonctions sur les additions est défini dans la classe Add. On a donc une forme de modularité organisée autour des types de données (voir figure 1.1).

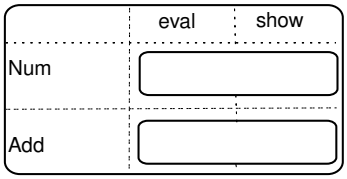


FIGURE 1.1: Décomposition de la structure d'un programme selon les types de données

Pour illustrer ceci, supposons qu'on voudrait ajouter un nouveau type de données Mult au type Expr. Pour ajouter le nouveau code, on crée une classe Mult et on y implémente les deux méthodes eval et show. Cette tâche de maintenance se fait dans un seul module et sans modifier les autres modules. On dit qu'elle est modulaire (voir figure 1.2).

Maintenant, supposons qu'on voudrait ajouter une nouvelle fonctionnalité au programme, par exemple une méthode derivation. Après avoir déclaré la méthode abstraite dans la classe Expr, on doit implémenter cette méthode dans toutes les sous classes de Expr. Ce type de maintenance est *transverse* ou *non-modulaire*.

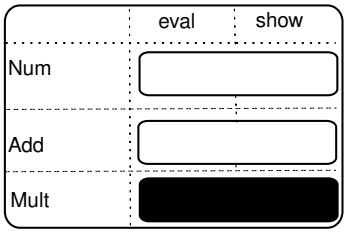


FIGURE 1.2: Maintenance modulaire sur un axe de décomposition d'un programme selon les types de données

Ici le programmeur doit étendre une classe par cas dans le type de données, alors que dans le cas de la maintenance modulaire, une seule classe était ajoutée. Cette maintenance transverse est donc plus coûteuse que la maintenance modulaire.

Dans cet exemple de programme, la décomposition principale correspond au type de données. Dans chaque classe, le code des différentes fonctions est structuré par méthodes, qui sont elles aussi des entités modulaires. On a donc un second degré de décomposition, ou de modularité.

Dans cette architecture, la modularité est favorisée sur l'axe de décomposition primaire, et, comme l'a montré l'exemple d'extension des fonctionnalités, défavorisée sur l'axe secondaire. C'est ce qu'on appelle la *tyrannie de la décomposition dominante* [TOHS99].

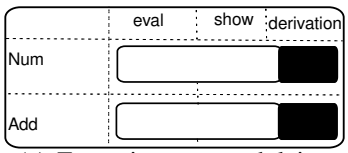
Notons qu'il ne suffit pas de choisir une bonne architecture pour se débarrasser de ce problème. Il n'existe pas d'architecture pour laquelle toutes les axes de maintenance seront modulaires.

La tyrannie de la décomposition dominante est un problème général qui empêche l'évolution modulaire (extension et maintenance modulaire) sur toutes les préoccupations d'intérêt. Ce problème se manifeste sous une autre forme plus spécifique qui est l'*Expression Problem*. L'*Expression Problem* a été formulé la première fois par Wadler [Wad98] : *« The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g no casts) »* . L'*Expression Problem* a été étudié dans nombreuses propositions dont la plupart s'appuient sur des spécificités d'un langage particulier [ZO05, LH06, Erl00, Bru03, CMLC06, Gar98, Gar00, KFF98, PJ98, Tor04, ZO01].

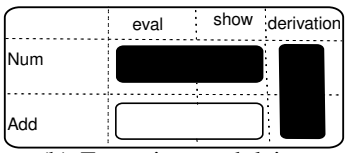
Les solutions proposées pour résoudre l'*Expression Problem* restent toujours dans le contexte de l'*extension modulaire* (ajout d'un cas dans le type de données ou ajout d'une fonction) et ne traitent pas la *maintenance modulaire* (modification d'un cas ou d'une fonction) [CD10].

La figure 1.3(a) schématise la décomposition de notre exemple de programme selon les types de données avec l'ajout d'une fonctionnalité derivation (extension non modulaire). La figure 1.3(b) montre que l'ajout de derivation devient modulaire si on applique les solutions proposées pour l'*Expression Problem*. La modification du comportement d'un des cas du type de données (comme remplacer les entiers par des flottants) doit être dispersée sur plusieurs modules (voir les parties colorées en noir sur la figure 1.3(c).

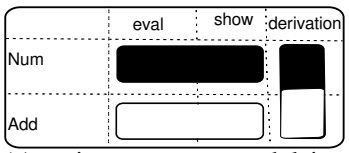
Ici l'extension est bien modulaire mais la maintenance ne peut plus l'être. On parle ici d'un cas de dégénérescence d'architecture.



(a) Extension non modulaire.



(b) Extension modulaire.



(c) maintenance non modulaire.

FIGURE 1.3: Extension et maintenance au sein de l'*Expression Problem*.

Les solutions proposées pour résoudre L'*Expression Problem* ne traitent donc pas le problème de la tyrannie de décomposition dominante dans sa généralité.

2 Solution proposée

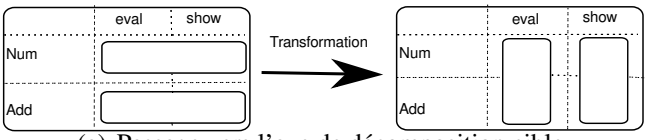
L'approche adoptée dans cette thèse consiste à traiter le problème de la tyrannie de la décomposition dominante pour rendre les maintenances modulaires. L'objectif est de pouvoir changer les axes de décomposition directement dans le code source pour que le programme s'adapte au type d'évolution voulue. Le scénario montré par la figure 1.4 illustre l'objectif de cette thèse :

1. Changer l'axe de décomposition de la structure du programme à maintenir en sorte que la tâche de maintenance voulue s'effectue d'une façon modulaire. Par exemple, l'ajout de la méthode dérivation au programme mentionné ci-dessus n'est pas conforme avec l'axe de décomposition selon les types de données. Ainsi pour que l'axe de décomposition soit conforme avec la tâche de maintenance, on transforme le programme vers une structure à axe de décomposition selon les fonctions (voir figure 1.4(a))

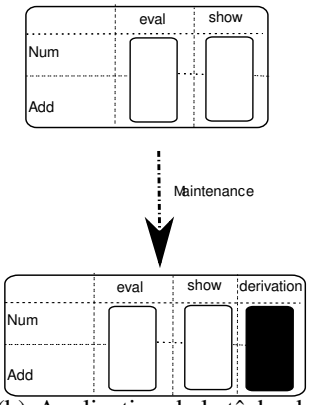
2. Une fois que le passage vers le bon axe de décomposition est fait, on effectue la tâche de maintenance d'une façon modulaire (voir figure 1.4(b)).

3. Après l'application de la tâche de maintenance modulaire dans l'axe de décomposition obtenu après la transformation, on applique le passage vers l'axe de décomposition du programme initial tout en générant le code ajouté dans ce programme (voir figure 1.4(c)).

La solution que nous proposons dans cette thèse permet à la fois d'effectuer différents types de maintenances d'une façon modulaire sur un même programme, éviter la dégénérescence et rester au sein du même langage (on n'a pas besoin de changer le paradigme pour décomposer le programme autrement comme par exemple le passage vers la programmation orienté aspect).



(a) Passage vers l'axe de décomposition cible.



(b) Application de la tâche de maintenance d'une façon modulaire.

(c) Génération du code ajouté par la maintenance dans le programme initial.

FIGURE 1.4: Un scénario illustrant une solution pour la tyrannie de la décomposition dominante.

Dans ce contexte nous cherchons à mettre en place des mécanismes qui permettent de transformer un programme vers diverses vues selon les tâches d'évolutions voulues pour que celles-ci restent toujours modulaires.

3 Contributions

Nous définissons des transformations de programmes qui permettent le passage de l'architecture d'un programme d'un axe de décomposition vers un autre. Nous implémentons ces transformations par la compo-

sition des opérations de refactoring disponibles dans les outils de refactoring. Nous les validons théoriquement par le calcul de la précondition minimale qui garantit statiquement leur bon déroulement. Nous les adaptons à des variations qui peuvent survenir dans les architectures en question et nous les validons sur une étude de cas réel. Nous proposons aussi un autre type de transformations qui porte plus sur le passage d'un niveau d'optimisation vers un autre et nous le validons.

Transformation entre patrons à propriétés complémentaires

Pour traiter le problème de la tyrannie de la décomposition dominante et l'*Expression problem*, nous ciblons deux patrons de conceptions à propriétés de modularité duales. Ces deux patrons sont le patron Composite et le patron Visiteur. Le passage entre ces deux patrons permet le changement de la propriété de modularité du programme ce qui lui offre deux axes de décomposition différents. L'automatisation et la réversibilité d'un tel passage permet au développeur de faire deux types de maintenances modulaires sur un même programme. Pour ceci, nous définissons deux algorithmes qui précisent la démarche à suivre pour passer de la structure Composite d'un programme vers sa structure Visiteur et vice-versa. Ces algorithmes se basent sur la composition des opérations de refactoring et leurs descriptions seront détaillées dans le chapitre 3.

Formalisation de la transformation

Les algorithmes que nous définissons pour la transformation réversible entre les patrons Composite et Visiteurs se basent sur la composition des opérations de refactoring. Dans le monde de refactoring, une opération de refactoring doit vérifier certaines préconditions pour qu'elle réussisse. Ceci n'est pas toujours vraie pour une séquence d'opérations de refactoring car les postconditions d'une opération peuvent être contradictoires par rapport aux préconditions de l'opération qui la suive, ce qui cause l'échec de la transformation. Pour éviter ceci, nous utilisons un système de calcul des préconditions de composition des opérations de refactoring. Notre contribution dans ce contexte se manifeste dans la définition de toutes les données nécessaires tels que des prédicats et des règles de calcul pour spécifier chaque opération de refactoring par rapport aux préconditions. Après la définition de ces données, nous les utilisons pour alimenter le système de calcul utilisé et nous générons ensuite la précondition minimale qui permet de garantir statiquement le bon déroulement de notre transformation en plus de sa validation. Cette contribution sera décrite et détaillée dans le chapitre 4.

Étude de l'impact des variations des patrons Composite et Visiteur sur la transformation

La transformation entre les patrons Composite et Visiteur mentionnée dans le section 1 est appliquée sur une version basique du patron Composite : des méthodes sans paramètres, sans types de retour, avec un seul niveau hiérarchique et avec une classe abstraite et non pas une interface. Après l'identification de quatre variations des deux patrons de conception en question, nous avons déduit que le changement de l'implémentation de ces deux patrons peut rendre cette transformation de base invalide. Les quatre variations que nous étudions sont : méthodes avec paramètres, méthodes avec types de retours différents, méthodes redéfinies aléatoirement sur plusieurs niveaux hiérarchiques et une interface au lieu d'une classe abstraite. Pour rendre la transformation de base valide sur ces quatre variations, nous adaptons les algorithmes de base et nous générons la précondition minimale qui assure la réussite de la transformation sur chaque variation (voir chapitre 5).

Validation de la transformation sur une étude de cas

Les quatre variations que nous étudions existent dans l'application JHotDraw [GI]. Nous orchestrons les différents algorithmes de transformation et nous les appliquons sur la structure Composite de cette application pour obtenir sa structure Visiteur avec la possibilité de revenir à la structure initiale. Nous appliquons aussi un scénario de maintenance modulaire sur la structure Visiteur obtenue de la transformation et grâce

à la transformation retour, le code ajouté au niveau du Visiteur serait répartie automatiquement aux bons endroits au niveau de la structure Composite. Une telle transformation est validée aussi par la génération de sa précondition minimale. La description de cette contribution est détaillée dans le chapitre 6.

Autre type de transformation

La bénéfice du passage entre deux patrons à propriétés duales, nous a poussé à définir un autre type de transformation qui cible la dualité optimisation/souplesse. Dans ce cadre, nous définissons une transformation qui permet d'introduire un patron Singleton dans un programme si on a besoin de l'optimisation et de le supprimer du même programme si on a besoin de la souplesse. Pour réaliser ceci, nous définissons des nouvelles opérations de refactoring qui ne sont pas encore implémentées par les outils de refactoring et nous générons aussi la précondition minimale qui valide cette transformation (voir chapitre 7).

4 Publications

1. **Akram Ajouli**, Julien Cohen, Jean-Claude Royer, *Transformations between Various Composite and Visitor implementations in Java*, 39th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2013). (Participation : 80 %)

2. Julien Cohen, **Akram Ajouli**, *Practical use of static composition of refactoring operations*, 28th ACM Symposium On Applied Computing, Portugal (2013). (Participation : 50 %)

3. **Akram Ajouli**, *An Automatic Reversible Transformation from Composite to Visitor in Java*, Conférence en IngénieriE du Logiciel (CIEL 2012) (Participation : 100 %)

4. Cohen Julien, Douence Rémi, **Ajouli AKram**, *An Invertible Program Restructurings for Continuing Modular Maintenance*, Software Maintenance and Reengineering (CSMR), 2012 16th European Conference, Szeged, Hungary. (Participation : 10 %)

2

État de l'art : vues et transformations de programmes

Sommaire

Dans ce chapitre, nous présentons tout d'abord des approches qui visent la résolution du problème de la tyrannie de décomposition dominante. Notamment, nous présentons quelques notions de vues de programmes qui montrent qu'un programme peut avoir plusieurs structures alternatives selon les attentes de l'utilisateur (compréhension, visualisation, maintenance...). Nous allons ensuite présenter les approches de transformation des programmes en tant qu'un mécanisme de passage d'une vue à l'autre ou d'une architecture à l'autre. En enfin, nous présentons quelques outils de refactoring qui peuvent être utilisés pour implémenter ces transformations.

1 Solutions à la tyrannie de la décomposition dominante

Nous faisons dans ce qui suit une classification de certaines notions de vues de programmes. Cette classification est faite selon la nature de la solution proposée.

Vues sur les données.

Wadler [Wad87] était parmi les premiers qui ont évoqué la notion de « vues » de programme. Il la considère comme un mécanisme qui permet aux types de données et aux structures de données d'être représentés sous des formes différentes. Ceci permet d'optimiser le parcours de certaines structures de données : par exemple une liste est construite de la tête vers la queue, ainsi pour la parcourir il faut suivre ce même sens. Ceci est intéressant par exemple si quelqu'un a besoin du premier élément d'une liste, mais s'il a besoin du dernier élément il doit parcourir toute la liste pour arriver jusqu'à la queue de la liste. Pour résoudre ceci Wadler propose de représenter une liste sous deux représentations : une représentation qui respecte le sens de construction standard d'une liste et l'autre qui respecte le sens inverse.

Cette approche est intéressante pour des finalités d'optimisation, mais moins intéressante pour la modularité de maintenance.

Modèles complémentaires

L'approche proposée dans [MR92] consiste à utiliser des représentations alternatives d'un même programme obtenues par analyse statique dans le but de faciliter la compréhension des programmes. Meyers et al. [MR92] appellent le mécanisme de représentations alternatives vues (*views*) : une vue selon eux est toute représentation qui visualise une partie d'un programme pour réduire sa complexité (exemples de vues proposées ici le graphe d'appels de fonctions et le graphe de dépendances entre les classes).

Une autre notion de vue a été présentée aussi dans [Saj00] et dans le même contexte. Elle consiste à fournir des représentations partielles alternatives pour un même programme comme les noms de ses variables, de ses fonctions...etc.

Ces deux approches aident à la compréhension des programmes et à l'augmentation de la performance des programmeurs. Bien que ceci est important, les deux approches ne s'adressent pas directement à ce que nous cherchons à résoudre (le problème de la tyrannie de la décomposition dominante).

Abstractions éditables obtenues par analyse statique

L'environnement de développement proposé dans [SI98] permet à l'utilisateur de voir des vues partielles du programme (variables, fonctions). Il s'agit d'écrire une transformation qui permet de générer la vue cible. Une transformation peut être une simple ligne de commande *grep* pour filtrer les fonctions d'un programme par exemple. L'environnement de développement proposé dans cette approche permet aussi à l'utilisateur d'éditer les vues de son programme. Les changements faits par l'utilisateur sur une vue seront appliquées automatiquement sur les autres vues.

Cette approche facilite la maintenance en permettant à l'utilisateur d'ajouter des vues éditables qui correspondent à ses besoins. Mais, chaque tâche de maintenance sur une vue, nécessite l'implémentation d'une transformation qui reflète les nouveaux changements sur la représentation du programme.

Décomposition des programmes selon des axes multiples

Tarr et al. [TOHS99] proposent un formalisme qui permet de modéliser les préoccupations existantes dans le programme par des entités appelées *hyperslices* : une *hyperslice* est un ensemble de modules qui participent à une même préoccupation unique. Ce formalisme sert à exprimer les préoccupations du programme tout le long de son cycle de vie de développement et non pas seulement au niveau du code.

L'intérêt de cette approche réside dans le fait qu'elle permet de séparer les préoccupations d'un programme, mais elle nécessite un apprentissage des concepts utilisés dans le formalisme proposé et elle propose des vues qui ne porte pas sur le même langage que celui du programme en question.

Descriptions complètes alternatives d'un programme

Black et Jones [BJ04] proposent une notion de vues de programmes sous le nom « vues multiples ». Les vues multiples sont des représentations alternatives qui peuvent accompagner un programme simultanément afin de montrer par exemple les dépendances entre les différentes classes ou entre les méthodes. Elles permettent aussi de faciliter la maintenance et la compréhension des logiciels. Ils proposent aussi des vues qui permettent de visualiser toutes les structures présentes dans un programme (y compris les classes, les interfaces, les paquetages, les méthodes, les attributs...) mais ces vues ne sont pas implémentées.

La représentation d'un programme sous plusieurs vues telles qu'elle est proposée dans [BJ04] est intéressante, mais elle a le même intérêt que celle faite par les graphes de dépendances qui sont fournis par la plupart des plateformes de développement. Cette approche concentre plus sur la compréhension des programmes que sur leurs maintenances.

Vues par aspect

L'idée globale des aspects consiste à encapsuler chaque préoccupation du programme dans un module qui est implémenté par un constructeur syntaxique qui s'appelle *aspect*. Kiczales et al. [KH01] sont les premiers qui ont défini l'approche de la programmation par aspects. Cette notion de vue a permis d'atténuer l'effet de la tyrannie de la décomposition dominante vue que les maintenances peuvent se faire d'une façon modulaire par rapport aux préoccupations du programme. Mais ces vues nécessitent soit l'apprentissage d'un nouveau langage soit la migration du programme à maintenir vers ce langage.

Bilan sur les vues

Ces notions de vue visent tous à représenter un programme sous plusieurs structures pour faciliter soit la compréhension, soit la maintenance. Mais, elles sont soit des vues partielles sur les programmes (or la tyrannie de la décomposition pourrait inclure tous les éléments du programme), soit elles nécessitent le changement du langage du code du programme ou le changement du niveau d'abstraction par rapport au programme en question.

2 Transformations de programmes

Nous avons vu dans la section précédente différentes notions de vues des programmes qui montrent des vues alternatives d'un programme. Nous nous concentrons dans cette thèse sur les vues en tant que structures alternatives d'un programme qui permettent d'y faire des maintenances modulaires. Nous présentons dans cette section quelques mécanismes permettant de passer d'une vue à l'autre.

Langages dédiés pour la transformation des programmes

Pour transformer les programmes, nous pouvons utiliser des langages dédiés (en anglais *DSL : Domain Specific Language*). Parmi ces langages, on trouve : Stratego [Vis00], JunGL [VEdM06], Arcum [SGL07], RASCAL [HKV12]. Le point commun entre ces langages est qu'ils se basent sur la réécriture du programme. Il permettent d'écrire des transformations complexes avec souplesse mais ils nécessitent un apprentissage et ne garantissent pas la sémantique lorsqu'ils sont utilisés uniquement pour la transformation. Ils sont aussi ad-hoc pour chaque évolution du programme.

Outils de refactoring disponibles

Selon Fowler [Fow99], la définition du refactoring est *"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure"*. Le refactoring est automatisé par l'implémentation de certains outils de refactoring. Il existe des outils de refactoring pour les langages objet et aussi pour les langages fonctionnels.

Parmi les premiers outils de refactoring dédiés pour les langages orienté objet, on trouve le REFACTORING BROWSER [RBJ97a] qui est utilisé pour effectuer des opérations de refactoring pour le langage SMALTALK. Un tel outil comprend les opérations de refactoring nécessaires pour manipuler la structure des programmes. L'environnement de développement ECLIPSE contient aussi un outil de refactoring riche en opérations de refactoring et doté d'une API pour ces opérations que nous pouvons les utiliser pour appeler ces opérations directement à partir d'un programme et qui nous permet aussi de les composer dans des séquences de refactoring. Un tel outil gratuit offre l'opportunité de construire des transformations complexes comme l'introduction des patrons de conception, mais reste moins rapide en matière d'évolution par rapport à IntelliJ IDEA avec ses deux versions payante et gratuite. IntelliJ IDEA intègre un outil de refactoring riche en catalogue d'opérations de refactoring avec une interaction dynamique avec les utilisateurs pour corriger les bogues ou bien prendre en compte des évolutions de certaines opérations.

D'autres outils de refactoring comportent une base théorique et formelle qui prouvent que certaines opérations parmi les opérations qu'il proposent sont correctes, comme le cas de HaRe l'outil de refactoring

du langage Haskell [LT05]. Un tel outil est efficace pour faire des compositions de refactoring pour changer l'architecture des programmes comme la transformation réversible proposé par Cohen et Douence [CD11] et qui permet de transformer un programme Haskell à structure types de données vers son équivalent à structure fonctions. Cet outil reste dépendant du paradigme fonctionnel, or nous nous intéressons dans cette thèse du paradigme orienté objet.

Pour voir des exemples d'utilisations de ces outils de refactoring ainsi qu'un bilan entre les opérations qu'ils proposent, plus de détails sont présentés dans l'annexe A.

Extraction des aspects

Kiczales et al. [KH01] sont les premiers qui ont défini l'approche de la programmation par aspects. L'idée globale des aspects consiste à encapsuler chaque préoccupation du programme dans un module qu'on l'appelle aspect. Les auteurs de cette approche ont défini toute un langage appelé AspectJ [KHH+01] pour programmer en aspect. Certains travaux ont choisi d'extraire les aspects à partir des programmes existants en utilisant le refactoring dans le but d'augmenter la modularité. Nous présentons ici quelques uns parmi eux.

Giunta et al. [GPT12] proposent une approche qui permet d'augmenter la modularité pour certains patrons de conception en les remplaçant par des aspects. Il s'agit de considérer l'implémentation du patron de conception comme une préoccupation et par la suite tout changement dans l'implémentation ou le remplacement par un autre patron n'influence pas la partie application du programme. Cette approche est intéressante pour augmenter la modularité des programmes mais elle se base sur la migration d'un programme orienté objet vers un programme orienté aspect ce qui peut compliquer la compréhension des patrons de conception surtout que certains patrons sont déjà difficiles à comprendre dans la paradigme orienté objet. Elle est intéressante aussi pour certains patrons comme le patron Composite qui se base sur la gestion des objets composites. Mais cette même approche peut être appliquée sans passer au paradigme aspect en transformant le patron Composite vers le patron monteur/*Builder*. Dans ce patron, la préoccupation gestion (création/-suppression) des objets composites est encapsulée dans une classe et séparée de la partie applicative du programme. Cette approche est aussi difficile à appliquer sur certains patrons où l'implémentation du patron est elle même une partie de l'application du programme comme par exemple le patron Visiteur où le double aiguillage représente à la fois une partie de l'implémentation du patron Visiteur et une partie qui assure le déroulement de l'application du programme.

D'autres approches comme [MMD07], [MF05], [HOU03], [BS02] et [BCH+06] ont visé la définition de certaines opérations de refactoring qui permettent de migrer les programmes orientés objet vers des programmes orientés aspect. Ceci est intéressant pour augmenter la modularité des programmes, mais la migration des programmes orientés objet existants vers leurs structures aspects peut être couteuse.

Bilan

Les langages dédiés permettent à la fois d'écrire la transformation d'un programme avec la tâche de maintenance voulue, mais ils ne garantissent pas la préservation de la sémantique du côté transformation. Le refactoring permet d'améliorer la structure d'un programme tout en préservant la sémantique, mais les outils de refactoring existants sont limités à des opérations de refactoring élémentaires et n'offrent pas une solution pour des transformations via les architectures connues dans l'ingénierie logiciel comme les patrons de conception par exemple. Et enfin, l'extraction des aspects permet de séparer les différentes préoccupations existantes dans un programme objet, mais avec le passage vers un nouveau langage qui est différent de celui du programme d'origine.

3 Transformation par composition des opérations de refactoring

Nous listons ici quelques travaux qui portent sur la composition des opérations de refactoring. Ce choix vient de fait que l'orchestration des opérations de refactoring élémentaires permet de construire des trans-

formations complexes pouvant assurer un passage automatique entre différente architectures d'un même programme.

Introduction des patrons de conception dans les programmes

Parmi les premiers travaux qui ont évoqué la transformation des programmes est celui de Takouda and Batory [BT95]. Dans cette approche, les auteurs proposent l'introduction des patrons de conception dans les programmes orientés objet. Leur approche se base sur la décomposition d'un patron de conception donné en une séquence de tâches de transformation et appliquer cette chaine à un programme initial pour atteindre à la fin la structure de ce patron. La transformation se fait sur le code source et ils offrent aussi des diagrammes qui permettent à l'utilisateur de capturer facilement les changements faits par chaque tâche de transformation. Cette approche est concrétisée par un outil qui automatise la transformation. L'introduction des patrons dans ces travaux est intéressante car elle est supportée par un outil qui l'automatise.

O'Cinnéide [OC00] propose aussi l'introduction des patrons de conception dans les programmes. Son approche se base sur la définition d'un ensemble de mini-transformations. Une mini-transformation est une tâche de transformation qui permet d'introduire une partie d'un patron donné dans un programme. L'introduction de chaque patron de conception se base sur l'orchestration d'une séquence de mini transformations. L'intérêt de cette approche se manifeste par le fait que d'une part, une mini-transformation peut être commune à plusieurs patrons à la fois, et d'autre part elle est automatisée et vérifiée par des préconditions qui garantissent statiquement la réussite de l'introduction d'un patron donné.

Ces deux travaux supportent l'introduction de plusieurs patrons. Mais, d'une part, les transformations proposées ne sont pas réversibles et d'autre part, ni l'un ni l'autre traite l'introduction du patron Visiteur. Le premier ne l'évoque pas et O'Cinnéide avoue que c'est difficile d'introduire ce patron (*"Overall Assessment : Impractical"* [OC00], page 151-152).

Transformation entre patrons de conception à propriétés complémentaires

Nous présentons ici les travaux qui traitent la transformation entre les programmes qui sont structurés selon le patron Composite et ceux qui sont structurés selon le patron Visiteur. Ces deux patrons qui décomposent les programmes selon deux axes complémentaires représentent le point essentiel dans cette thèse.

Introduction du patron Visiteur dans une simple hiérarchie de classes. Pour illustrer chaque approche dans ce contexte, nous considérons le programme Java 1 comme un exemple d'une hiérarchie de classes. Ce programme est constitué d'une classe abstraite Graphic qui a deux sous classes Ellipse et Square et deux méthodes print et show. Nous considérons aussi le programme 2 comme le programme qui lui est équivalent et qui est structuré selon le Visiteur.

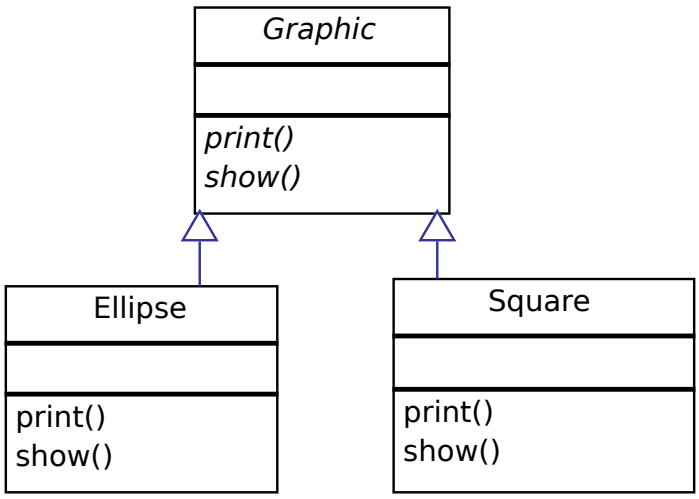
Parmi les premiers travaux qui ont évoqué l'introduction du patron Visiteur dans un programme est celui de Roberts et al. [RBJ97b]. L'algorithme 1 montre la démarche adoptée dans cette approche pour obtenir le programme 2.

Le deuxième algorithme dédié à l'introduction du Visiteur est celui de Mens et Tourwé [MT04]. Il s'agit d'un pseudo algorithme qui permet de transformer une hiérarchie de classes vers un patron Visiteur. Pour expliquer cette approche nous considérons toujours le programme Java 1. L'objectif de la transformation proposée par Mens et Tourwé [MT04] est d'obtenir le programme 2.

Le programme 2 est obtenu par l'application du pseudo algorithme 2 sur le programme 1. Plusieurs étapes de l'algorithme réfèrent à des opérations de refactoring qui sont définies par Fowler [Fow99]. Nous détaillons cette approche ici car elle représente la base de départ de notre transformation Composite vers Visiteur qui sera détaillée dans le chapitre 3.

L'application des étapes 1 et 2 de l'algorithme 2 transforme le programme 1 comme suit :

```
class Ellipse extends Graphic{
 void print(PrintVisitor v){
   v.visit(this);
```




```
abstract class Graphic {

    abstract public void print();

    abstract public void show();

}
```

```
class Square extends Graphic {

    int i;

    public void print() {
        System.out.print("Square(" + i + ")");
    }
    public void show(){
        System.out.print("Square.");
    }
}
```

```
class Ellipse extends Graphic{

    int i1, i2;

    public void print() {
        System.out.print("Ellipse: (" + i1 +"," + i2 + ")");
    }
    public void show(){
        System.out.print("Ellipse.");
    }
}
```

Programme 1: Programme Java avec une simple hiérarchie de classes.

```
abstract class Graphic {

    public void print() {
        accept(new PrintVisitor());
    }
    public void show() {
        accept(new ShowVisitor());
    }
    public abstract void accept(Visitor v);

}
```

```
class Square extends Graphic {

    int i;

    public void accept(Visitor v) {
        v.visit(this);
    }
}
```

```
class Ellipse extends Graphic{

    int i1, i2;

    public void accept(Visitor v) {
        v.visit(this);
    }
}
```

```
public abstract class Visitor {

  public abstract void visit(Square s);

  public abstract void visit(Ellipse e);
}
```

```
public class PrintVisitor extends Visitor {

  public void visit(Square s) {
    System.out.print("Square(" + s.i + ")");
  }
  public void visit(Ellipse e) {
    System.out.print("Ellipse: (" + e.i1 +"," + e.i2 + ")");
  }
}
```

```
public class PrettyPrintVisitor extends Visitor {

  public void visit(Square s){
    System.out.print("Square.");
  }
  public void visit(Ellipse e){
    System.out.print("Ellipse.");
  }
}
```

Programme 2: Structure Visiteur du programme de la figure 1

1. Créer la hiérarchie Visiteur avec sa racine (l'interface du Visiteur) : cette hiérarchie contient les classes Visitor, PrintVisitor et ShowVisitor.
2. Ajouter dans la hiérarchie Visiteur une méthode *visit* pour chaque sous classe du programme 1. Les méthodes sont visit(PrintVisitor v) et visit(ShowVisitor v).
3. Créer un déléguée pour chaque méthode métier.
4. Ajouter un paramètre du type Visiteur pour chaque méthode déléguée. Par exemple, ajouter un paramètre de type PrintVisitor pour le déléguée de la méthode print.
5. Déplacer le code de chaque méthode déléguée vers la méthode *visit* correspondante.
6. Renommer toutes les méthodes déléguées en accept.

Algorithme 1: Algorithme de transformation d'une hiérarchie de classes vers un patron Visiteur (Roberts et al. [RBJ97b]).

1. Déplacement des quatre définitions des méthodes print et show vers respectivement deux classes qui prennent les noms de ces deux méthodes.
2. Renommage des méthodes déplacées vers les classes PrintVisitor et ShowVisitor par visit
3. Création d'une classe abstraite Visitor comme une classe mère des classes PrintVisitor et ShowVisitor
4. Ajout des déclarations abstraites des méthodes *visit* dans la classe Visitor.
5. Ajout d'une méthode *accept* dans les deux sous classes de la classe Graphic par l'extraction des corps des méthodesprint et show.
6. Créer une délégation entre les méthodes print et show et la méthode *accept* dans la classe Graphic

Algorithme 2: Algorithme de transformation d'une hiérarchie de classes vers un patron Visiteur (Mens and Tourwé [MT04]).

```
 }
 void show(ShowVisitor v){
   v.visit(this);
 }
}
class PrintVisitor {

 void visit(Ellipse e) {
    System.out.print("Ellipse: (" + e.i1 +"," + e.i2 + ")");
  }
 ...
}
class ShowVisitor {

 void visit(Ellipse e) {
    System.out.print("Ellipse.");
  }
 ...
}
```

Par l'application des étapes 3 et 4, le programme 1 se change comme suit :

```
abstract class Visitor{
 abstract visit(Ellipse e);
 abstract visit(Square s);
}

class ShowVisitor extends Visitor{...}
class PrintVisitor extends Visitor{...}
```

Après l'application des étapes 5 et 6 nous obtenons le programme 2.

Un autre algorithme qui apparait après l'approche [MT04] est celui de Kerievsky [Ker04] et qui suit la même démarche de celui de Roberts et al. [RBJ97b]. Mais Kerievsky n'applique pas la partie de l'étape 1 qui sert à créer une interface pour la partie Visiteur.

Le point commun entre ces trois algorithmes est :

- Il ne traitent pas le cas où la hiérarchie de classe comporte la récursion des structures de données (caractéristique d'un vrai Composite),
- Ils ne sont pas réversible.
- Ils ne sont pas automatisés.
- Ils ne sont pas formalisés.

L'algorithme de Mens et al. [RBJ97b] est plus explicite que les deux autres et il représente un point de départ pour la transformation entre un Composite et un Visiteur. Les deux autres algorithmes sont similaires sauf que celui de Kerievsky [Ker04] introduit un Visiteur sans interface ce qui impose de créer une méthode *accept* pour chaque Visiteur.

La transformation décrite ci-dessus était automatisée, formalisée et définie comme réversible par Cohen et Douence [CD10] mais dans le paradigme fonctionnel et plus précisément le langage Haskell ce qui ne nous intéresse pas dans cette thèse.

Transformation d'un interpréteur vers un Visiteur. Hills et al. [HKVDSV11] ont développé un outil qui permet de transformer une instance d'un Visiteur vers une instance d'Interpréteur. Leur outil a été appliqué sur un programme réel (leur propre outil de transformation Rascal [KSV09]). Cette approche est aussi proche de la notre mais elle ne fournit pas une transformation réversible et elle n'est pas formalisée.

Variations des instances Composite et Visiteur. Les patrons de conception n'ont pas une seule instance, mais ils peuvent varier d'implémentation. Ces variations peuvent influencer la transformation définie pour une telle instance. Selon notre connaissance il y a une seule approche qui traite ce sujet qui est l'approche de Kerievsky [Ker04]. Nous avons déjà mentionné que Kerievsky [Ker04] propose un algorithme qui transforme une hiérarchie de classes vers un Visiteur classique. Nous présentons ici une autre variation traité par Kerievsky [Ker04] (chapitre 10, page 320).

Nous considérons le programme 3 comme un programme d'entrée pour la transformation. Dans ce programme, la méthode extractNames sert à parcourir chaque objet de la structure de données graphics pour collecter les noms de ces objets qui sont sauvegardés dans la variable d'instance name de chaque objet. L'objectif de la transformation proposée par Kerievsky [Ker04] est de déléguer la collecte du résultat au patron Visiteur pour ne pas tenir compte du type de l'objet à parcourir. Pour ceci Kerievsky [Ker04] définit une démarche qui permet d'obtenir le programme 4. Nous n'allons pas détailler sa démarche ici vu que la structure du Visiteur considérée ici est plus intéressante pour des raisons d'optimisation (éviter les conversions de types). Dans cette variation, nous pouvons aussi éviter d'introduire le patron Visiteur en créant des déclarations abstraites du champs name dans l'interface Graphic et faisant un parcours récursif sur la collection d'objets du type de cette racine à partir de la méthode extractNames.

```
public interface Graphic {

}
```

```
public class Ellipse implements Graphic {
    String name;
    Ellipse(String name){
        this.name = name;
    }
}
```

```
public class Square  implements  Graphic{
     String name;
     Square(String name){
         this.name = name;
     }

}
```

```
public class NamesExtractor {
     StringBuffer names =  new StringBuffer();

     List<Graphic> graphics = new ArrayList<Graphic>();

    String extractNames(){
        for(Graphic g : graphics){
            if(g instanceof Ellipse){
                names.append(((Ellipse) g).name);
            }
            if(g instanceof Square){
                names.append(((Square) g).name);
            }
        }
      return names.toString();

    }
}
```

Programme 3: Une hiérarchie de classe parcourue par une méthode de collecte de résultats.

Nous mentionnons d'autres travaux qui présentent des implémentations de ces deux patrons sans qu'ils étudient l'influence de ces implémentations sur la transformation entre ces deux patrons. le patron Composite est présenté dans [GHJV95] sous une seule implémentation. La seule discussion faite sur l'implémentation de ce patron concerne l'emplacement des méthodes de gestion des composites (comme *add* et *remove*) ce qui ne nous intéresse pas vu que nous nous concentrons plus sur les méthodes métier ou les méthodes qui sont intéressantes pour le rôle du programme en question. Gamma et al. [GHJV95] présente

```
public interface Graphic {
 abstract void accept(AccumulateVisitor v);
}
```

```
public class Ellipse implements Graphic {
 String name;
 Ellipse(String name){
  this.name = name;
 }
}
```

```
public class Square  implements Graphic {
  String name;
  Square(String name){
   this.name = name;
  }
  public void accept(AccumulateVisitor v){
   v.visit(this);
  }
}
```

```
public interface AccumulateVisitor {
 abstract  void visit(Ellipse e);
 abstract  void visit(Square s);
}
```

```
public class NamesExtractor implements AccumulateVisitor{
 StringBuffer names =  new StringBuffer();

 List<Graphic> graphics = new ArrayList<Graphic>();

 public void visit(Ellipse e){
  names.append(e.name);
 }
 public void visit(Square s){
  names.append(s.name);
 }
 String extractNames(){
  for(Graphic g : graphics){
     g.accept(this);
  }
  return names.toString();
 }
}
```

Programme 4: Structure Visiteur du programme 3

aussi une seule implémentation du Visiteur. Buchlovsky and Thielecke [BT06] appelle l'implémentation proposée dans [GHJV95] un Visiteur interne (le parcours de la hiérarchie du Composite se fait à l'intérieur de la hiérarchie elle même). Buchlovsky and Thielecke [BT06] propose aussi l'implémentation d'un Visiteur externe dans laquelle le parcours de la hiérarchie du Composite se fait à l'extérieure de la hiérarchie (dans la partie Visiteur).

4 Outillage pour la composition d'opérations de refactoring

La composition des opérations de refactoring consiste à appliquer une séquence d'opérations de refactoring comme une seule macro opération de refactoring. Cette composition peut être construite de deux manières :

- Une composition non statique qui concentre plus sur l'adaptation du refactoring aux attentes de l'utilisateur plutôt que de vérifier si la composition est valide ou non.
- Une composition statique qui concentre plus sur la validité de la composition avant son application. Ceci se fait par la génération des préconditions de la totalité de la composition par différent systèmes de calcul que nous allons décrire dans la deuxième partie de cette section.

Composition non statique

Li et Thompson [LT12] proposent un langage dédié pour composer les opérations de refactoring de Wrangler (outil de refactoring pour le langage Erlang). Ce langage est écrit en Erlang et offre à l'utilisateur la possibilité d'écrire ses propres opérations de refactoring en composant des opérations élémentaires qui existent déjà dans l'outil de refactoring. Chaque opération de refactoring élémentaire est représenté dans ce langage dédié par une commande qui permet à l'utilisateur d'enter les bons arguments. L'utilisateur peut choisir si la composition se lance en mode atomique ou bien interactive. Le deuxième mode lui permet de répéter l'exécution de la même opération avec des arguments différents. Ce langage permet d'effectuer des macro opérations de refactoring sans avoir besoin d'implémenter de zéro des nouvelles opérations. Mais la composition des opérations de refactoring dans cette approche ne peut être vérifiée qu'elle est valide qu'après sa réussite. Ceci est du au fait que ce travail ne traite pas la génération des préconditions de la totalité de la composition mais il se contente de la réussite de chaque opération de refactoring comme si elle est exécutée toute seule. La vérification se fait directement par l'outil de refactoring. Ceci peut être couteux si les opérations dépendent les unes des autres et que l'une d'entre elle échoue au milieu de l'exécution de la composition.

VaKilian et al. [VCM+13] proposent une approche qui se base sur la décomposition des opérations qui sont complexes vers des opérations plus atomiques. Ces opérations atomiques peuvent alors être composées manuellement par l'utilisateur étape par étape d'une manière interactive. Ceci est justifié par le fait que l'automatisation de l'application d'une composition d'opérations peut causer des erreurs sans que l'utilisateur se rend compte. Un autre argument pour éviter l'application d'une composition de refactoring d'une façon atomique, c'est que cette composition nécessite une inférence de ses préconditions par contre son application étape par étape permet de vérifier chaque opération toute seule en plus de la vérification faite par l'utilisateur après chaque changement. Cette approche vise à décomposer des opérations de refactoring complexes afin de permettre à l'utilisateur de pouvoir bien maîtriser la transformation qu'il veut faire et pour avoir plus de souplesse à utiliser les mini-opérations dans d'autres compositions de refactoring.

L'utilisation de la composition non statique des opérations de refactoring est intéressante lorsque l'exécution de chaque opération est indépendante de l'autre vu qu'on est pas besoin d'inférer les préconditions de la composition. Mais dès que l'exécution de la composition des opérations de refactoring devient atomique et que chaque opération dépend de celle qui la précède, on doit valider la composition avant son lancement pour éviter l'échec de la transformation au milieu de son exécution et le risque de perdre l'architecture du programme initial ou les erreurs imprévus qui peuvent survenir. Pour prévoir la réussite d'une composition d'opérations de refactoring, certains travaux peuvent calculer la précondition minimale pour ces transformations. C'est ce qu'on appelle la composition statique d'opérations de refactoring.

Composition statique

La composition statique des opérations de refactoring consiste à la validation de la composition avant son exécution. Supposons par exemple qu'on veut ajouter une nouvelle classe A, puis nous voulons ensuite renommer cette classe par B. Nous appliquons ainsi la composition d'opérations de refactoring suivante :

CreateClass(A) ; *RenameClass(A,B)*

En fait, chaque opération de refactoring impose un certain nombre de préconditions pour qu'elle réussisse. Par exemple pour ajouter une classe A, on doit vérifier que la classe n'existe pas. La réussite d'une opération de refactoring élémentaire dépend de la satisfaction de ses propres préconditions, mais lorsque on applique une suite d'opérations de refactoring, on doit tenir compte des changements faits par chaque opération pour pouvoir définir les préconditions de cette suite d'opérations. Voici ci-dessous une spécification très simplifiée de chacune de ces deux opérations :

- L'opération *CreateClass(A)* doit vérifier que la classe A n'existe pas : $\neg\texttt{ExistsClass}(A)$.
- L'opération *RenameClass(A,B)* doit vérifier que la classe A existe ($\texttt{ExistsClass}(A)$) et que la classe B n'existe pas ($\neg\texttt{ExistsClass}(B)$).

La précondition de cette composition est $\neg\texttt{ExistsClass}(A) \wedge \neg\texttt{ExistsClass}(B)$. Nous remarquons que la précondition $\texttt{ExistsClass}(A)$ qui est demandée par l'opération *RenameClass(A,B)* ne figure pas dans la précondition de la composition. Ceci est du à l'inférence de cette précondition car elle est garantit par l'opération *CreateClass(A)*.

Nous présentons deux systèmes de calcul de préconditions différents pour les compositions des opérations de refactoring :

Calcul en avant des préconditions

Cette approche est proposée par O'Cinnéide [OC00]. Elle consiste tout d'abord à identifier les préconditions et les postconditions de chaque opération, puis le calcul des préconditions de la composition commence par la vérification des préconditions de chaque opération qui sont garanties par l'opération qui la précède. Toute précondition garantie par une postcondition d'une opération est retirée de la précondition de la composition comme c'est le cas pour l'exemple précédent où la précondition $\texttt{ExistsClass}(A)$ est déduite automatiquement vu qu'elle est garantie par l'opération *CreateClass(A)* qui va changer la précondition $\neg\texttt{ExistsClass}(A)$ en une postcondition $\texttt{ExistsClass}(A)$. On peut imaginer dans ce contexte une fonction f, qui décrit les postconditions de chaque opération de refactoring, appelée *forward description*, et qui sera utilisée comme suit :

- Pour l'opération *CreateClass(A)*, on a $f_{\textit{CreateClass(A)}}(\neg\texttt{ExistsClass}(A)) = \texttt{ExistsClass}(A)$
- Pour l'opération *RenameClass(A,B)*, on a $f_{\textit{RenameClass(A,B)}}(\neg\texttt{ExistsClass}(B)) = \texttt{ExistsClass}(B)$

Les préconditions de la première opération seront conjointes avec les préconditions de la deuxième opération qui ne sont pas garanties par les postconditions de la première. Par exemple, on va faire la conjonction suivante : $\neg\texttt{ExistsClass}(A) \wedge \neg\texttt{ExistsClass}(B)$. S'il n'y a pas une contradiction dans cette conjonction alors la séquence des opérations de refactoring correspondante est légale sinon la séquence n'est pas légale. Cette vérification sera faite à partir de la première opération jusqu'à arriver à la dernière pour générer à la fin la précondition qui garantit qu'une telle composition est valide ou non avant son lancement.

Calcul en arrière des préconditions

Kniesel et Koch [KK04] proposent une autre approche de calcul similaire à la précédente. Ils proposent de décrire les opérations de refactoring par une fonction b (on l'appelle *backward description* ou rétro-description) qui étant donné une condition qui doit être vérifiée après l'application de l'opération en question, génère la condition correspondante qui doit être vérifiée avant l'application de l'opération. Par exemple, avec l'opération *RenameClass(A,B)*, les rétro-descriptions sont présentées comme suit :

- $b_{RenameClass(A,B)}(\texttt{ExistsClass}(B)) = \texttt{ExistsClass}(A)$: si la classe A existait avant l'application de l'opération de refactoring, alors la classe B existera forcément après l'application de l'opération en question.
- $b_{RenameClass(A,B)}(\texttt{ExistsClass}(A)) = FALSE$ si la classe A existait avant, alors elle ne vas pas exister après l'application de l'opération en question.
- $b_{RenameClass(A,B)}(\texttt{ExistsMethod}(B, m)) = \texttt{ExistsMethod}(A, m)$: pour toute méthode m qui existait avant dans la classe A, alors elle va exister dans la classe B.

Les rétro-descriptions d'une opération peuvent être représentées de la façon suivante aussi :
$b_{RenameClass(A,B)}$:
$\texttt{ExistsClass}(A) \mapsto FALSE$
$\texttt{ExistsClass}(B) \mapsto \texttt{ExistsClass}(A)$
$\texttt{ExistsMethod}(B, m) \mapsto \texttt{ExistsMethod}(A, m)$

Chaque opération est décrite par un ensemble de préconditions et un ensemble de rétro-descriptions. Le calcul des préconditions d'une séquence de n opérations de refactoring se fait selon la définition suivante : Pour une conjonction d'une séquence de paires d'opérations de refactoring associées à leurs préconditions qu'on le note $COp_{1..n>=2}$ avec C est la précondition, on définit la contribution $Contrib_k^j$ de COp_j par rapport à la précondition de $COp_{k<=j}$ ($k = 1..n-1$) comme suit :

$$\begin{array}{|l|}\hline Contrib_j^j := C_j \text{ pour } j = 1..n \\ Contrib_k^n := b_{Op_k} \circ b_{Op_{k+1}} \circ ... \circ b_{Op_{n-1}}(C_n) \text{ pour } k = 1..j-1, j = 2..n \\ \hline\end{array}$$

Pour comprendre cette formule, on va l'appliquer sur notre exemple. On a deux opérations dont la première est d'indice $n = 1$ qui est l'opération *CreateClass(A)* et la deuxième opération dans la séquence a l'indice $n = 2$ et qui est *RenameClass(A,B)*, en appliquant la formule on aura :

$$\boxed{Contrib_1^2 = Contrib_{CreateClass(A)}^{RenameClass(A,B)} := b_{CreateClass(A)}(C_{RenameClass(A,B)})}$$

Pour calculer le résultat, on doit avoir tout d'abord les rétro-descriptions de l'opération *CreateClass(A)* :
$b_{CreateClass(A)} := \texttt{ExistsClass}(A) \mapsto TRUE$

En on fournit les conditions de l'opération *RenameClass(A,B)*, par exemple :
$\texttt{ExistsClass}(A)$
$\wedge \neg\texttt{ExistsClass}(B)$

Le résultat de calcul est :
$Contrib_{CreateClass(A)}^{RenameClass(A,B)} = b_{CreateClass(A)}(C_{RenameClass(A,B)})$
$= TRUE$
$\wedge \neg\texttt{ExistsClass}(B)$

$= \neg\texttt{ExistsClass}(B)$

Pour générer la précondition minimale de la composition selon cette approche, on doit maintenant faire la conjonction des contributions suivantes (avec $n = 2$ pour l'exemple ci-dessus) :

$$\begin{array}{|l|}\hline Contrib_j^j := C_j \ pour j = 1..n \\ Contrib_1^1 \wedge Contrib_1^2 = Contrib_{CreateClass(A)}^{CreateClass(A)} \wedge Contrib_{CreateClass(A)}^{RenameClass(A,B)} = \\ C_{CreateClass(A)} \wedge Contrib_{CreateClass(A)}^{RenameClass(A,B)} \\ \hline\end{array}$$

On a déjà le résultat de $Contrib_{CreateClass(A)}^{RenameClass(A,B)}$ qui est $\neg\texttt{ExistsClass}(B)$ et on a la précondition de l'opération *CreateClass(A)* (notée ici $C_{CreateClass(A)}$) qui est $\neg\texttt{ExistsClass}(A)$, d'où le résultat final et la précondition minimale de la composition est :

$\neg\texttt{ExistsClass}(A)$
$\wedge \neg\texttt{ExistsClass}(B)$

Cet exemple montre comment le calcul en arrière des préconditions en utilisant les rétro-descriptions permet d'éliminer les préconditions d'une opération qui sont garanties par les rétro-descriptions de l'opération précédente (exemple : $\texttt{ExistsClass}(A)$).

Discussion. La différence entre les deux systèmes, c'est que le premier se base sur l'identification des préconditions pour générer les postconditions, alors que le deuxième se base sur la spécification des rétro-descriptions pour générer la précondition minimale qui garantit la réussite d'une séquence d'opérations de refactoring.

Kniesel et Koch [KK04] ont décrit environ 33 opérations de refactoring par leurs précondition et rétro-descriptions en utilisant 46 propositions, par contre O'Cinnéide [OC00] a décrit 13 opérations de refactoring à l'aide de 36 propositions.

Nous verrons dans le chapitre 4 comment nous avons exploité l'approche de Kniesel et Koch [KK04] pour générer la liste des préconditions minimales qui permettent de valider les transformations que nous proposons dans cette thèse.

D'autres travaux ciblent aussi la composition de refactoring comme [Mar04] qui composent les opérations de refactoring mais au niveau des modèles, or nous nous intéressons au code source.

5 Bilan

Dans ce chapitre, nous avons présenté tout d'abord quelques notions de vues de programmes qui montrent qu'un programme peut avoir plusieurs structures alternatives selon les attentes de l'utilisateur (compréhension, visualisation, maintenance...). Nous avons aussi présenté la transformation des programmes en tant qu'un mécanisme de passage d'une vue à l'autre et nous nous sommes concentré sur la transformation entre les patrons de conception en tant que transformation entre les vues qui pourrait être une solution pour la tyrannie de la décomposition dominante. Nous avons aussi exploré quelques systèmes de calcul qui permettent de générer la précondition minimale qui garantit la réussite d'une composition d'opérations de refactoring d'une façon statique.

3

Transformation entre un patron Composite et un patron Visiteur

Sommaire

Dans ce chapitre nous proposons une transformation automatique, réversible et qui préserve la sémantique entre un programme Java structuré selon le patron de conception Composite et son équivalent structuré selon le patron Visiteur. Cette transformation offre à un programme deux structures alternatives dont chacune respecte un axe de décomposition et un type de maintenance modulaire. Nous rappelons dans ce chapitre ces deux patrons et les spécificités de chacun vis-à-vis de la modularité, puis nous décrivons la transformation que nous avons définie pour passer de l'un à l'autre.

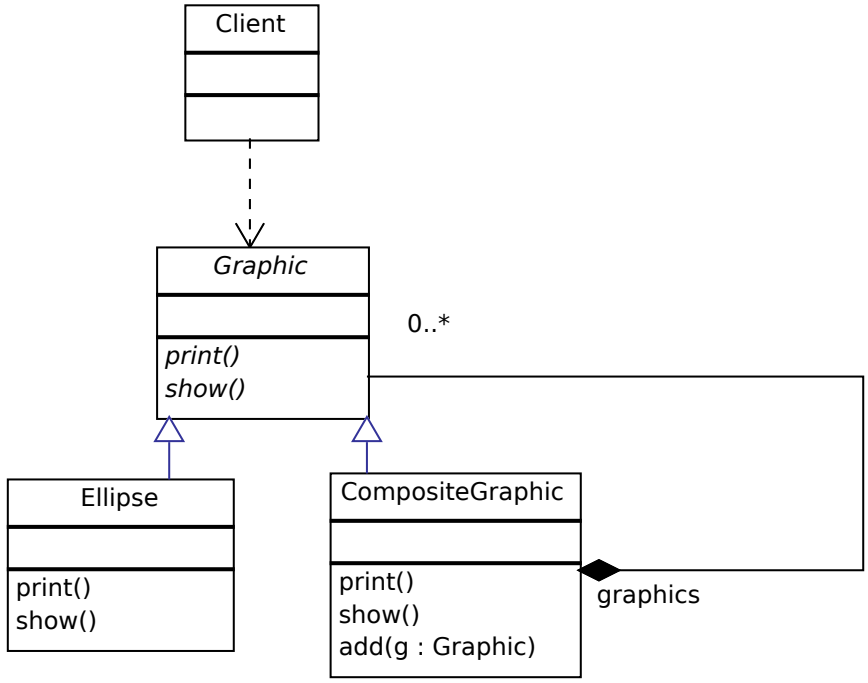
1 Le patron de conception Composite

Caractéristiques

Le rôle d'un patron Composite selon Gamma et al. [GHJV95] est de composer des objets en des structures arborescentes pour représenter des hiérarchies composant/composé.

Le patron Composite est caractérisé par une racine et un ensemble de composants dont l'une d'eux est utilisé pour contenir, ajouter, supprimer et gérer des objets composants. Par exemple dans le programme 5, la classe Graphic joue le rôle de la racine du Composite, les deux classes Ellipse et Composite représentent les composants. La classe Composite joue aussi le rôle d'un conteneur des objets du Composite. Ceci est fait par le stockage des objets composants dans la structure de données graphics qui peut être déclarée dans cette classe comme suit :

```
List<Graphic> graphics = new ArrayList<Graphic>;
```




```
abstract class Graphic {
    abstract public void print();

    abstract public void show();
}
```

```
class Ellipse extends Graphic{

     public void print() {
        System.out.println("Ellipse :" + this);
     }
     public void show(){
        System.out.println("Ellipse corresponding to the object " + this + ".");
      }
}
```

```
class Composite extends Graphic {
    List<Graphic> graphics = new ArrayList<Graphic>();

   public void print() {
      System.out.println("Composite:");
      for (Graphic g : graphics) {g.print();}
   }
   public void show(){
      System.out.println("Composite " + this + " composed of:");
      for (Graphic g : graphics) {g.show();}

      System.out.println("(end of composite)");
   }
   public void add(Graphic g) {
      graphics.add(g);
   }}
```

Programme 5: Un Programme structuré selon le patron de conception Composite

La manipulation des objets composants se fait par la définition des méthodes suivantes dans la classe Composite :

```
public void add(Graphic g){
  graphics.add(g);
}


public void remove(Graphic g){
  graphics.remove(g);
}
```

Gamma et al. [GHJV95] considèrent les méthodes add et remove comme des méthodes de gestion dans le patron Composite. Dans cette thèse nous nous intéressons uniquement à ce que nous appelons méthodes métier (voir définition ci-dessous).

Méthode métier. Une méthode métier dans ce contexte est toute méthode représentant une fonctionnalité du programme. Nous pouvons répartir les méthodes du programme 5 selon leur rôle :

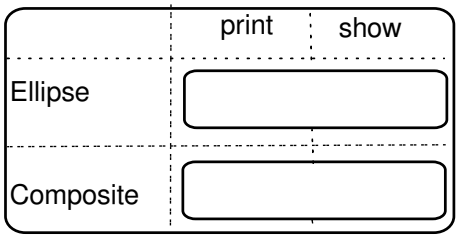
Méthode	Rôle
print	métier
show	métier
add	gestion

Un Composite est caractérisé aussi par la récursion qui se manifeste dans la façon dont les méthodes métier sont invoquées par les objets composants dans la classe qui est chargée de les parcourir. Dans le programme 5, la classe Composite comporte des appels récursifs des méthodes print et show :

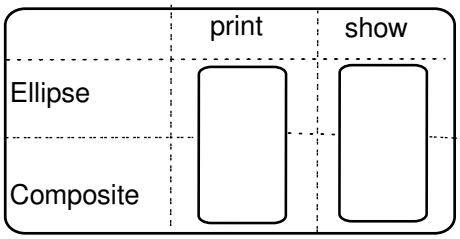
```
abstract class Graphic{
   abstract print();
   abstract show();
}

class Composite extends Graphic{
   public void print(){
    for(Graphic g : graphics){
         g.print();
    }
   }
   public void show(){
    for(Graphic g : graphics){
         g.show();
    }
   }
```

Axe de modularité dans le patron Composite

Dans le programme 5 structuré selon le patron Composite, chaque sous classe de la racine Graphic contient le code des deux méthodes métier print et show. Donc, le code de chaque méthode métier est dispersé sur des classes/modules différents. L'encapsulation du code des deux méthodes métiers par classe donne au programme une décomposition selon les types de données. La figure 3.1(a) montre visuellement que le programme 5 est décomposé selon les types de données Ellipse et Composite. Par contre, nous remarquons que le code de chaque méthode métier est dispersé sur ces deux types de données.



(a) Axes de décomposition d'un patron Composite.



(b) Axes de décomposition d'un patron Visiteur.

FIGURE 3.1: Décomposition d'un programme selon deux axes différents.

L'axe de décomposition par types de données qui caractérise ce programme favorise un type de maintenance par rapport à d'autres. En effet, l'ajout d'un type de donnée dans un programme structuré selon une instance de Composite est une tâche de maintenance modulaire. Par exemple, si le développeur veut ajouter un graphique de type rectangle dans le programme 5, il crée une sous classe Rectangle de la classe Graphic, puis, il implémente les deux méthodes print et show dans ce module sans toucher aux autres modules. Maintenant, si le développeur veut ajouter une nouvelle fonction/méthode métier, il doit toucher à plusieurs classes ou modules. Par exemple s'il veut enrichir le programme ci-dessus par une nouvelle fonctionnalité qui fait la coloration des graphiques, il déclare une méthode abstraite color(int color) dans la classe Graphic, puis, il doit implémenter cette méthode dans toutes les sous classes ce qui rend la maintenance non modulaire.

La figure 3.2 montre que l'ajout d'un type de donnée (un module Rectangle) dans un programme structuré selon le patron Composite s'effectue d'une façon modulaire, par contre l'ajout d'une méthode métier nécessite la dispersion de son code sur les différents modules.

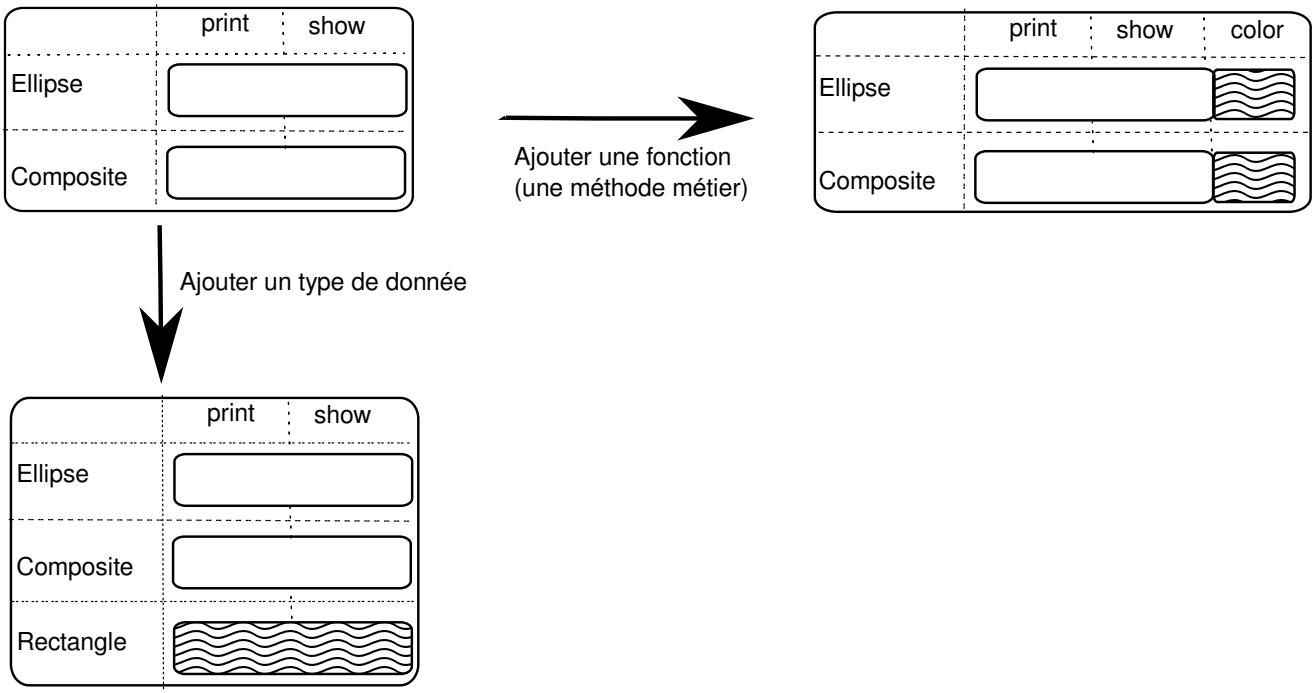


FIGURE 3.2: Maintenances modulaires et d'autres non modulaires dans un programme structuré selon le patron Composite.

Le patron de conception Composite favorise l'axe de décomposition selon les types de données et permet de faire des maintenances modulaires selon cette axe de décomposition par rapport à d'autres.

2 Le patron de conception Visiteur

Caractéristiques

Le rôle d'un patron Visiteur selon Gamma et al. [GHJV95] est de *« faire la représentation d'une opération applicable aux éléments d'une structure d'objets. Il permet de définir une nouvelle opération, sans qu'il soit nécessaire de modifier la classe des éléments sur lesquels elle agit. »*

Le programme 6 est structuré selon le patron Visiteur. Les constituants les plus intéressants dans ce programme sont :

La classe abstraite Visitor. Elle contient deux méthodes visit(Composite c) et visit(Ellipse e) dont les signatures indiquent chacune quelle classe est visitée. Dans ce programme, on a deux classes à visiter : Composite et Ellipse. La visite de ces deux classes est effectuée par l'invocation des méthodes *visit* par un objet de type Visitor dans la méthode accept qui est définie dans chacune de ces classes comme suit :

```
public accept(Visitor v){
 v.visit(this);
}
```

Les classes concrètes PrintVisitor et ShowVisitor. Les méthodes *visit* déclarées abstraites dans la classe Visitor sont implémentées dans ces deux classes. La méthode visit(Ellipse e) contient le code qui correspond aux objets de type Ellipse et la méthode visit(Composite c) contient le code qui correspond aux objets de type Composite.

La classe abstraite Graphic. La présence de la méthode abstraite accept dans cette classe permet aux visiteurs d'accéder aux classes Composite et Ellipse à partir de cette classe. Ceci se fait par l'invocation de la méthode *accept* par les objets de type Graphic qui sont stockées dans la collection graphics comme c'est montré par le code suivant :

```
class PrintVisitor extends Visitor{
  public void visit(Composite c) {
  ...
  for (Graphic g : c.graphics) {g.accept(this);}
 }
 ...
}
```

Les classes Composite et Ellipse . Elles représentent les nœuds à visiter par les classes PrintVisitor et ShowVisitor. Chacune de ces deux classes implémente la méthode accept déclarée dans la classe Graphic :

```
public accept(Visitor v){
 v.visit(this);
}
```

Cette implémentation permet à chaque nœud d'accepter un objet de type Visitor qui invoque la méthode *visit* en y passant un objet de type de ce nœud (représenté par this). Ceci permet de préciser à la classe visiteur quel objet elle visite.

Nous déduisons que chaque visiteur visite les nœuds à l'aide de l'invocation de la méthode *accept* par les objets composants stockés dans la structure de données garphics. Cette méthode une fois invoquée elle va déterminer le nœud à visiter. Ceci permet d'accéder au code de la méthode *accept* du nœud correspondant qui est v.visit(this). Une fois le code de la méthode *accept* accédé, l'appel v.visit(this) permet de déterminer le composant à visiter à l'aide de this. Par exemple, si le visiteur est la classe PrintVisitor et si l'appel v.visit(this) est effectué dans la méthode *accept* définie dans la classe Ellipse, le code qui va être exécuté est celui de la méthode visit(Ellipse e) :

```
class PrintVisitor extends Visitor{
  ...
  public void visit(Ellipse e) {
   System.out.println("Ellipse :" + e);}
}
```

Remarque. L'exécution de la méthode *accept* dépend en même temps du type du Visiteur et du type de composant, c'est ce qu'on appelle double aiguillage (en anglais, on l'appelle *double dispatch*). Le double aiguillage est parmi les caractéristiques clés du patron Visiteur.

Axe de modularité dans le patron Visiteur

Dans le programme 6 structuré selon le patron Visiteur, le code métier qui correspond à chacune des méthodes print et show est encapsulé chacun dans un seul module : le code correspondant à la méthode print est encapsulé dans la classe PrintVisitor et le code correspondant à la méthode show est encapsulé dans la classe ShowVisitor. Ceci donne au programme une décomposition selon les fonctions. La figure 3.1(b) montre visuellement que le programme 6 est décomposé selon les fonctions/méthodes print et show.

L'axe de décomposition par fonctions de ce programme favorise une maintenance modulaire selon les fonctions. Par exemple, si le développeur veut ajouter une fonctionnalité de coloration (comme on a vu dans la section 1), il crée une sous classe ColorVisitor de la classe Visitor, puis il implémente les méthodes *visit* déclarées dans cette classe. Par exemple, le code qui permet de colorer un objet de type Ellipse est implémenté dans le corps de la méthode visit(Ellipse e). Dans ce même programme, si le développeur veut ajouter une nouvelle classe Rectangle dans la partie Composite du patron Visiteur, il déclare une méthode abstraite *visit* dans la classe Visitor, puis il implémente cette méthode dans les sous classes du Visiteur ce qui rend la maintenance non modulaire.

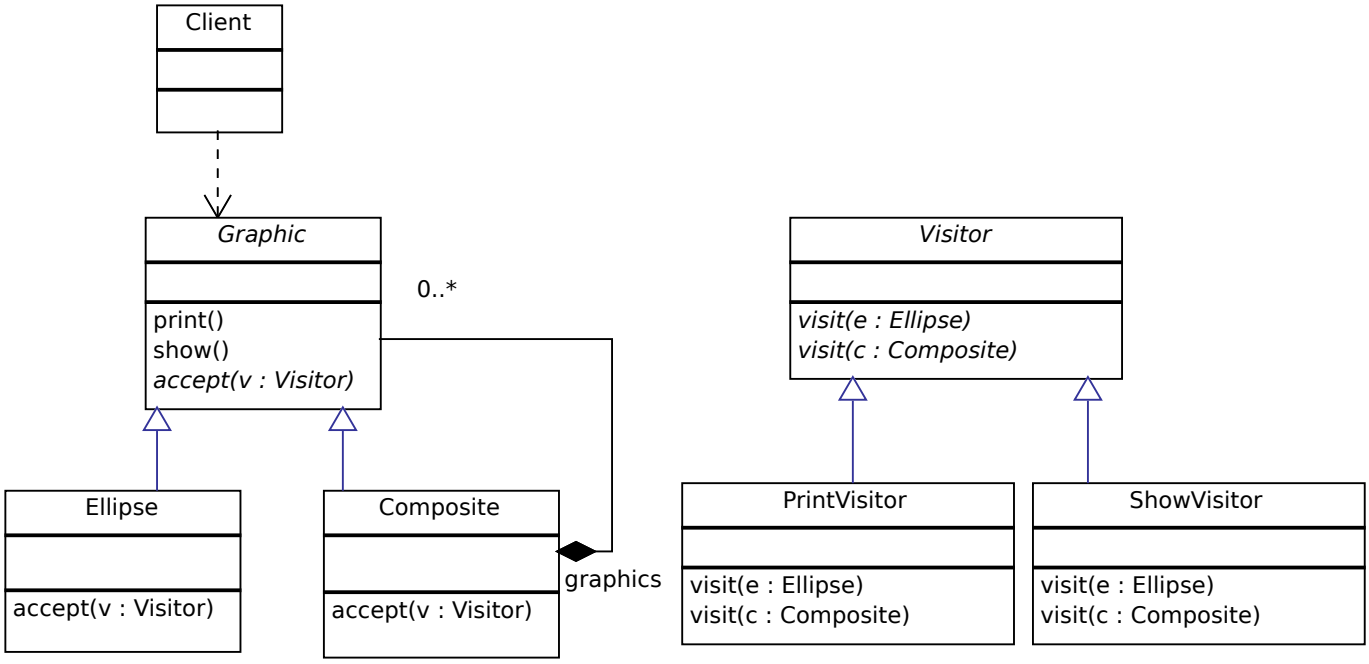
La figure 3.3 montre que l'ajout d'une fonction dans un programme structuré selon le patron Visiteur s'effectue d'une façon modulaire, par contre l'ajout d'un type de donnée est une tâche de maintenance non modulaire.

Le patron de conception Visiteur favorise l'axe de décomposition selon les fonctions et permet de faire des maintenances modulaires selon cette axe de décomposition par rapport à d'autres.

3 Dualité entre le patron Composite et Visiteur

Nous remarquons que les patrons de conception Composite et Visiteur montrent deux axes de décompositions duales. Tandis que le premier permet à un programme d'être décomposé selon les types de données, le deuxième lui permet d'être décomposé selon les fonctions (voir figure 3.1). Cette dualité montre aussi que les deux patrons offrent deux types de maintenances modulaires complémentaires. Le patron Composite permet d'effectuer des maintenances modulaires selon les types de données, par contre le patron Visiteur les permet selon les fonctions.

La dualité entre les deux patrons montre l'intérêt de passer d'un patron à l'autre afin de bénéficier de deux types de maintenances modulaires différents sur un seul programme.




```
abstract class Graphic {

 public void print() {accept(new PrintVisitor());}

 public void show() {accept(new ShowVisitor());}

 public abstract void accept(Visitor v);}
```

```
class Ellipse extends Graphic{

  public void accept(Visitor v) {v.visit(this);}}
```

```
class Composite extends Graphic {
 List<Graphic> graphics = new ArrayList<Graphic>();

 public void accept(Visitor v) {v.visit(this);}}
```

```
public abstract class Visitor {

 public abstract void visit(Ellipse e);

 public abstract void visit(Composite c);}
```

```
public class PrintVisitor extends Visitor {

 public void visit(Composite c) {
  System.out.println("Composite:");
  for (Graphic g : c.graphics) {g.accept(this);}
 }
 public void visit(Ellipse e) {
  System.out.println("Ellipse :" + e);
 }}
```

```
public class ShowVisitor extends Visitor {

 public void visit(Composite c) {
  System.out.println("Composite " + c + " composed of:");
  for (Graphic g : c.graphics) { g.accept(this);}

  System.out.println("(end of composite)");
 }
 public void visit(Ellipse ellipse) {
  System.out.println("Ellipse corresponding to the object " + e + ".");
 }}
```

Programme 6: Un Programme structuré selon le patron de conception Visiteur

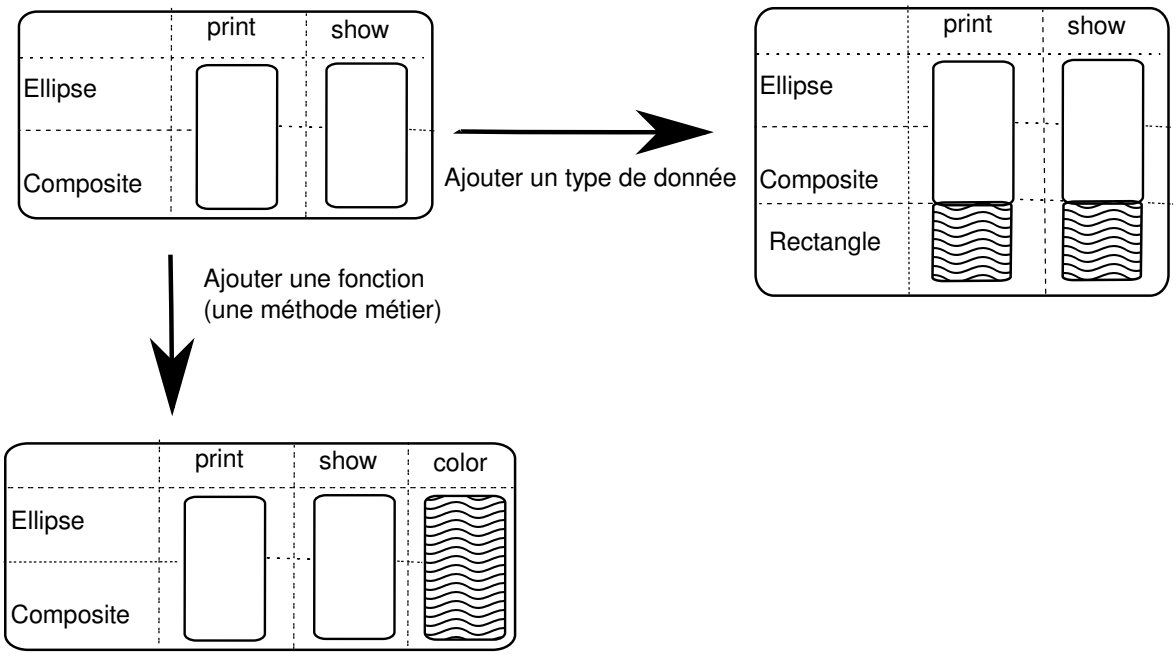


FIGURE 3.3: Maintenances modulaires et d'autres non modulaires dans un programme structuré selon le patron Visiteur.

Pour bénéficier des deux axes de décomposition présentés ci-dessus et ainsi des deux types de maintenances modulaires, nous montrons dans ce qui suit comment nous automatisons le passage réversible entre les deux patrons tout en préservant la sémantique.

4 Transformation d'une hiérarchie de classes vers un Visiteur (État de l'art)

Comme indiqué dans le chapitre 2, il y a des travaux qui ont été fait sur la transformation des patrons. Parmi les approches les plus proches de la notre on trouve celle de Mens et Tourwé [MT04]. Ils transforment une hiérarchie de classe vers un Visiteur. Cette hiérarchie de classe est représentée par le programme Java 1 (page 22) qui comporte une classe abstraite Graphic, deux sous classes Square et Ellipse et deux méthodes (print et show) implémentées dans les sous classes.

L'algorithme 2 de transformation proposé par Mens et Tourwé [MT04] (page 24) représente un point de départ pour la transformation du Composite vers le patron Visiteur. Les points suivants ne sont pas traités dans la transformation proposée par Mens et Tourwé [MT04] :

1. La récursion qui caractérise le patron Composite (la récursion est déjà expliqué dans la section 1).
2. L'automatisation de l'algorithme de transformation par la formalisation de chaque étape de l'algorithme et son implémentation.
3. La réversibilité de la transformation.
4. La génération de la précondition qui garantit la réussite de la transformation (chapitre 4).
5. L'extension de la transformation pour prendre en compte quelques variations dans les patrons Composite et Visiteur (chapitre 5).

Les trois premiers points seront traités dans la section 5, tandis que, les deux derniers points seront traités plus tard dans les chapitres 4 et 5

5 Transformation réversible entre les patrons Composite et Visiteur

Nous montrons dans cette section comment automatiser une transformation d'une instance d'un patron Composite vers un patron Visiteur et vice versa. L'instance du Composite que nous traitons représente une instance simple de ce patron : méthodes sans paramètres, sans types de retour, dont la racine est une classe abstraite, avec un seul niveau hiérarchique et dont les méthodes métier sont définies dans toutes les sous-classes (des autres variations de ce patron sont traitées dans le chapitre 5). L'instance du Visiteur que nous traitons ici est un Visiteur externe dans lequel le parcours des nœuds se fait à partir de la partie Visiteur et non pas à partir de la partie Composite.

A titre d'exemple, nous partons du programme 7 qui est structuré selon le patron Composite et nous voulons obtenir le programme 8 et qui respecte la structure du patron Visiteur. Nous présentons tout d'abord quelques définitions préliminaires, puis, nous détaillons les étapes de refactoring automatisant le passage d'un programme à l'autre et vice-versa.

```
abstract class Graphic {
    abstract public void print();

    abstract public void show();
}
```

```
class Ellipse extends Graphic{

     public void print() {
        System.out.println("Ellipse :" + this);
     }
     public void show(){
        System.out.println("Ellipse corresponding to the object " + this + ".");
      }
}
```

```
class Composite extends Graphic {
    List<Graphic> graphics = new ArrayList<Graphic>();

   public void print() {
      System.out.println("Composite:");
      for (Graphic g : graphics) {g.print();}
   }
   public void show(){
      System.out.println("Composite " + this + " composed of:");
      for (Graphic g : graphics) {g.show();}

      System.out.println("(end of composite)");
   }
   public void add(Graphic g) {
      graphics.add(g);
   }}
```

Programme 7: Programme initial (structure Composite)

Définitions préliminaires et notations

Refactoring. *« Refactoring is the process of changing software system in such a way that it does not alter the external behavior of the code »* Fowler [Fow99].

Opération de refactoring élémentaire. Une opération de refactoring élémentaire utilisée dans nos algorithmes de transformation est une opération qui se compose d'une seule opération de l'outil de refactoring utilisé.

Automatisation de l'utilisation d'une opération de refactoring élémentaire. Il s'agit d'utiliser l'opération à partir de l'API de l'outil de refactoring et non pas du GUI de l'outil de refactoring.

```
abstract class Graphic {

 public void print() {accept(new PrintVisitor());}

 public void show() {accept(new ShowVisitor());}

 public abstract void accept(Visitor v);}
```

```
class Ellipse extends Graphic{

  public void accept(Visitor v) {v.visit(this);}}
```

```
class Composite extends Graphic {
 List<Graphic> graphics = new ArrayList<Graphic>();

 public void accept(Visitor v) {v.visit(this);}}
```

```
public abstract class Visitor {

 public abstract void visit(Ellipse e);

 public abstract void visit(Composite c);}
```

```
public class PrintVisitor extends Visitor {

 public void visit(Composite c) {
  System.out.println("Composite:");
  for (Graphic g : c.graphics) {g.accept(this);}
 }
 public void visit(Ellipse e) {
  System.out.println("Ellipse :" + e);
 }}
```

```
public class ShowVisitor extends Visitor {

 public void visit(Composite c) {
  System.out.println("Composite " + c + " composed of:");
  for (Graphic g : c.graphics) { g.accept(this);}

  System.out.println("(end of composite)");
 }
 public void visit(Ellipse ellipse) {
  System.out.println("Ellipse corresponding to the object " + e + ".");
 }}
```

Programme 8: Programme structuré selon le patron Visiteur et équivalent au programme 7

Par exemple, si on veut déplacer trois méthodes m1, m2 et m3 de la classe A vers la classe B, on va abstraire l'opération *MoveMethod* en définissant un ensemble de méthodes M et on définit cette opération comme suit :

ForAll m in M do
MoveMethod(A,m,B)

Ceci permet d'éviter d'utiliser cette opération trois fois à partir de la GUI de l'outil de refactoring.

Chaine d'opérations de refactoring. Nous appelons chaine d'opérations la liste des appels d'une opération de refactoring appliquée automatiquement. Par exemple, la chaine d'occurrences de l'opération élémentaire *MoveMethod* sera représentée comme suit :

1. *MoveMethod(A,m1,B)*
2. *MoveMethod(A,m2,B)*
3. *MoveMethod(A,m3,B)*

Nous pouvons aussi représenter une chaine d'opérations sous la forme d'une liste :

[*MoveMethod(A,m1,B)* **;**
MoveMethod(A,m2,B) **;**
MoveMethod(A,m3,B)]

Opération de refactoring composée. Nous appelons une opération de refactoring composée dans notre transformation toute opération qui comporte un ensemble d'opérations de refactoring élémentaires.

Par exemple, pour appliquer l'exemple précédent tout en renommant les méthodes dans la classe cible, on définit une seule opération *MoveAndRenameMethod* qui se base sur l'enchainement des deux opérations *MoveMethod* et *Rename*. Pour définir cette opération, on définit une fonction $genname$ qui génère un nom pour chaque méthode. L'opération prend la forme suivante :

ForAll m in M do
MoveAndRenameMethod(A,m,B,$genname(m)$)

Toutes les opérations de refactoring élémentaires et composées que nous avons définies et utilisées sont détaillées dans l'annexe B.

Notations

Nous avons vu que pour automatiser les opérations de refactoring offertes par les outils de refactoring, nous formalisons les paramètres de ces opérations par des notations (par exemple noter l'ensemble de méthodes ou de classes en question). Le tableau suivant présente les notations que nous allons utiliser dans les algorithmes de transformation 3 (page 44) et 4 (page 49) :

Notation	Définition	Exemple
$\mathbb{M}$	Ensemble de méthodes métier	$\mathbb{M}$ ={print, show}
$\mathbb{C}$	Ensemble de classes de la hiérarchie du Composite à l'exception de la racine	$\mathbb{C}$ ={Ellipse, Composite}
S	Racine de la structure Composite	S = Graphic
vis	Fonction qui donne le nom de la classe Visiteur associée à la méthode métier correspondante	vis(print) = PrintVisitor, vis(show) = ShowVisitor
$\mathbb{V}$	Ensemble de classes Visiteurs, $\mathbb{V} = \{vis(m)\}_{m \in \mathbb{M}}$	$\mathbb{V}$ = {PrintVisitor, ShowVisitor}
aux	Fonction utilisée pour générer des noms temporaires qui représentent les méthodes métier correspondantes, ces noms sont utilisés uniquement dans les étapes intermédiaires	aux(print) =printAux, aux(show) =showAux

Nous décrivons et détaillons maintenant les algorithmes de transformation que nous avons définis pour automatiser le passage du Composite vers le Visiteur et vice-versa.

Transformation Composite → Visiteur

L'algorithme 3 permet de passer d'un programme structuré selon une instance du patron Composite vers programme structuré selon une instance du patron Visiteur. Nous rappelons ici que l'instance du Composite que nous traitons dans ce chapitre est une instance avec méthodes sans paramètres, sans types de retour, dont la racine est une classe abstraite, avec un seul niveau hiérarchique et dont les méthodes métier sont définies dans toutes les sous classes.

1) ForAll m in $\mathbb{M}$ do *CreateEmptyClass*(vis(m))
2) ForAll m in $\mathbb{M}$ do *CreateIndirectionInSuperClass*(S,m, aux(m))
3) ForAll m in $\mathbb{M}$, c in $\mathbb{C}$ do *InlineMethodInvocations*(c, m, aux(m))
4) ForAll m in $\mathbb{M}$ do *AddParameterWithReuse*(S, aux(m), vis(m))
5) ForAll m in $\mathbb{M}$, c in $\mathbb{C}$ do *MoveMethodWithDelegate*(c, aux(m), vis(m), "visit")
6) *ExtractSuperClass*($\mathbb{V}$, "Visitor")
7) ForAll m in $\mathbb{M}$ do *UseSuperType*(S, aux(m), vis(m), "Visitor")
8) *MergeDuplicateMethods*(S, {aux(m) }$_{m \in \mathbb{M}}$, "accept")

Algorithme 3: Algorithme de base pour la transformation Composite→Visiteur

Les étapes de l'algorithme 3 peuvent être classées selon trois phases principales :

Préparation du déplacement du code métier (étapes de 1 à 4). Il s'agit de fournir tout ce qui permet de déplacer le code métier de la partie Composite vers la partie Visiteur pour permettre à l'outil de refactoring de déplacer ce code.

Déplacement du code métier (étape 5). Il s'agit de déplacer le code métier vers la partie Visiteur.

Retouches finales (étapes de 6 à 8). Il s'agit d'effectuer les retouches finales pour obtenir la structure standard du Visiteur.

Préparation du déplacement du code métier vers la partie Visiteur.

Étape 1. On doit tout d'abord créer les classes Visiteurs qui vont recevoir plus tard le code métier qui leurs correspond. En appliquant cette étape sue le programme 7, l'opération *CreateEmptyClass* va créer deux classes PrintVisitor et ShowVisitor pour recevoir plus tard les codes métier respectivement des méthodes print et show.

Étape 2. Pour garder la même interface utilisateur, on crée des déléguées pour les méthodes métiers. Sur l'exemple, on va créer deux méthodes déléguées pour les méthodes print et show comme suit :

```
abstract class Graphic{

    public void print() {
      printAux();
    }
    public abstract printAux();

    public void show() {
      showAux();
    }
    public abstract showAux();
}

class Composite extends Graphic{

  public void printAux() {
        System.out.print("Composite: " + this + " with: (");
        for (Graphic g : graphics) {
            g.print();
        }
        System.out.println(")");
   }
   public void showAux() {
        System.out.println("Composite " + this + " composed of:");
        for (Graphic g : graphics) {
            g.show();
        }
        System.out.println("(end of composite)");
  }
}
```

Nous remarquons qu'après l'application de cette étape, la méthode métier print qui était récursive devient indirectement récursive dans la classe Composite (même chose pour la méthode show), ceci est due à la création de la délégation pour une méthode récursive. Les méthodes auxiliaires obtenues (qui portent les noms temporaires) vont devenir plus tard des méthodes *accept*.

Étape 3. Nous remarquons que dans le programme résultant de l'étape 2, les méthodes printAux et showAux ne sont pas récursives ce qui va influencer la création d'un appel récursif de la méthode *accept* dans la boucle for plus tard. Pour résoudre ce problème, inliner les appels de print et show dans la classe Composite respectivement vers printAux et showAUx, obtenant ainsi des méthodes auxiliaires récursives :

```
abstract class Graphic{
```

```
    public void print() {
      printAux();
    }
    public abstract printAux();

   ...
}

class Composite extends Graphic{
  public void printAux() {
        System.out.print("Composite: " + this + " with: (");
        for (Graphic g : graphics) {
            g.printAux();
        }
        System.out.println(")");
    }
    ...
}
```

Étape 4. Pour déplacer une méthode (étape 5), on doit préciser sa destination. La destination d'une méthode doit être indiquée dans la liste de ses paramètres pour l'opération *Move* de IntelliJ IDEA. Pour traiter ceci, nous utilisons l'opération AddParameterWithReuse qui va ajouter pour chaque méthode auxiliaire un paramètre de type de la classe de destination. Dans notre exemple, les méthodes auxiliaires vont avoir la signature suivante : printAux(PrintVisitor v) et showAux(showVisitor v) dans chaque classe de la hiérarchie Composite.

Après l'application de l'étape 4, les méthodes auxiliaires sont prêtes à être déplacées vers leurs classes Visiteurs correspondants. Ces méthodes sont les déléguées des méthodes métier et par la suite le code métier qui est déléguée vers ces méthodes va être déplacé vers la partie Visiteur. Voici ci-dessous une idée sur l'état du programme à ce stade de transformation :

```
abstract class Graphic{
   public void print(){
       printAux(new PrintVisitor());
   }
   public abstract void printAux(PrintVisitor v);

   ...
}


class Composite extends Graphic{
   public void printAux(PrintVisitor v){
      System.out.println("Composite:");
      for (Graphic g : graphics) {
            g.printAux(new PrintVisitor());
      }
   }
  ...
}
```

Déplacement du code métier vers la partie Visiteur

Étape 5. Pour créer le double aiguillage entre la partie Visiteur et la partie Composite, le code de chaque méthode auxiliaire est déplacée vers sa classe Visiteur correspondante tout en déléguant ce code à la méthode déplacée. En effet, la méthode auxiliaire reste dans sa classe d'origine et la méthode déléguée prend

le nom *visit*. Ceci s'effectue à l'aide de l'opération *MoveMethodWithDelegate* et voici ci-dessous une partie du code qui montre l'état du programme à ce stade :

```
class Composite extends Graphic{

   public void printAux(PrintVisitor v){
      v.visit(this);
   }
   ...
}
```

```
class PrintVisitor {
   public void visit(Composite c){

     System.out.print("Composite: " + c + " with: (");
        for (Graphic g : c.graphics) {
            g.printAux();
        }
        System.out.println(")");

   }
   ...
}
```

Retouches finales pour l'obtention de la structure Visiteur

Étape 6. Les méthodes *visit* créées dans l'étape précédente et qui sont des déléguées des méthodes auxiliaires ont chacune une même signature sur les deux classes Visiteurs comme suit :

```
class PrintVisitor{
   public void visit (Ellipse e){...}

   public void visit (Composite c){...}
}
```

```
class ShowVisitor{
   public void visit (Ellipse e){...}

   public void visit (Composite c){...}
}
```

Pour créer une interface qui communique avec la partie Composite, on doit créer une classe mère abstraite pour les deux classes Visiteurs et créer des déclarations abstraites des deux méthodes *visit*. Pour ceci, nous utilisons l'opération *ExtractSuperForvisitors* qui va extraire d'une part une classe mère pour les classes Visiteurs, et d'autre par elle va créer des déclarations abstraites des deux méthodes *visit* de la façon suivante :

```
 abstract class visitor {

   public abstract void visit (Ellipse e);

   public abstract void visit (Composite c);
 }

 class PrintVisitor extends Visitor{...}

 class ShowVisitor extends Visitor{...}
```

Étape 7. Pour accomplir la création du double aiguillage entre la partie Visiteur et Composite, nous changeons les types des paramètres des méthodes auxiliaires par le type de la racine du Visiteur. Ceci impose l'accès aux méthodes *visit* par l'interface de la partie Visiteur. Nous utilisons l'opération *GeneraliseParameter* pour réaliser cette étape. Ainsi les méthodes auxiliaires changent de signature comme suit :

```
abstract class Graphic{

  public void print(){
    printAux(new PrintVisitor());
  }
  abstract printAux(Visitor v);
  ...
}

class Ellipse extends Graphic{
  public void printAux (visitor v){
    v.visit(this);
  }
  ...
}
```

Étape 8. A ce stade de la transformation, le programme contient des méthodes auxiliaires qui ont des corps équivalents :

```
class Ellipse extends Graphic{
  public void printAux (Visitor v){
    v.visit(this);
  }

  public void showAux (Visitor v){
    v.visit(this);
  }
}
```

Nous allons remplacer ces deux méthodes par une seule méthode qui s'appelle *accept*. Ceci est légal vu que les deux méthodes printAux et showAux ont le même comportement. Pour créer une seule méthode *accept* qui remplace ces deux méthodes nous utilisons l'opération *MergeDuplicateMethods* (cette opération est une opération composée et elle est détaillée dans le chapitre 4 et dans l'annexe B).

Résultat.

Après l'application de cette transformation nous obtenons le programme 8 qui respecte la structure du patron Visiteur avec quelques changement dans le *layout* du programme. Cette transformation automatique prend à peu près 4 secondes pour transformer le programme de la figure 7.

Cette transformation est automatisé par l'implémentation des opérations de refactoring qui la composent avec IntelliJ IDEA. Elle est utilisée par un simple clic à partir du menu de refactoring et elle est déployée sous la forme d'un *plugin* IntelliJ IDEA.

Transformation Visiteur → Composite

L'algorithme 4 permet de passer d'un patron Visiteur vers un patron Composite et récupérer ainsi le programme 7.

Les étapes de l'algorithme 4 sont aussi classées selon trois phases principales :

I) ForAll v in $\mathbb{V}$ do AddSpecializedMethodInHierarchy(S,"accept","Visitor",v)

II) *DeleteMethodInHierarchy*(S,accept,"Visitor")

III) ForAll c in $\mathbb{C}$ do *PushDownAll*("Visitor","visit",c)

IV) ForAll v in $\mathbb{V}$, c in $\mathbb{C}$ do *InlineMethod*(v,"visit",c)

V) ForAll m in $\mathbb{M}$ do *RenameMethod*(S,accept,vis(m),aux(m))

VI) ForAll m in $\mathbb{M}$ do *RemoveParameter*(S,aux(m),vis(m))

VII) ForAll m in $\mathbb{M}$ do *ReplaceMethodDuplication*(S,m)

VIII) ForAll m in $\mathbb{M}$ do *PushDownImplementation*(S,m)

IX) ForAll m in $\mathbb{M}$ do *PushDownAll*(S,aux(m))

X) ForAll m in $\mathbb{M}$, c in $\mathbb{C}$ do *InlineMethod*(c,aux(m))

XI) ForAll v in $\mathbb{V}$ do *DeleteClass*(v)

XII) *DeleteClass*("Visitor")

Algorithme 4: Algorithme de base pour la transformation Visiteur → Composite

Préparation du déplacement du code métier vers la partie Composite (étapes de I à III). Il s'agit de fournir tout ce qui permet de déplacer le code métier de la partie Visiteur vers la partie Composite pour permettre à l'outil de refactoring de déplacer ce code.

Déplacement du code métier (étape IV). Il s'agit de déplacer le code métier vers la partie Visiteur.

Retouches finales (étapes de V à XII). Il s'agit d'effectuer des tâches de retouches nécessaires pour récupérer la structure Composite du programme.

Préparation du déplacement du code métier vers la partie Composite.

Étape I. Le patron Visiteur comporte des appels des méthodes *accept* qui représentent des délégateurs des méthodes *visit* dans sa partie Composite :

```
public void accept(Visitor v){
    v.visit(this);
}
```

Pour déplacer le code métier de la partie Visiteur vers la partie Composite, nous devons faire un *inline* pour les appels des méthodes *visit* dans la partie Composite. Mais ici, nous ne pouvons pas effectuer ce refactoring pour deux raisons :

- L'appel de la méthode *visit* à cet état du programme ne permet pas de préciser vers quel code métier cet appel est déléguée.
- L'appel se réfère vers la déclaration abstraite des méthodes *visit*.

Le premier problème est traité dans cette étape par l'application de l'opération *AddSpecializedMethodInHierarchy* et l'opération *DeleteMethodInHierarchy* (étape II) et nous traitons le deuxième problème dans l'étape III. L'opération *AddSpecializedMethodInHierarchy* est utilisé pour générer à partir de la méthode *accept* initiale deux autres méthodes *accept* dans la partie Composite de la façon suivante :

```
abstract class Graphic{
    ...
    public abstract void accept(Visitor v);

    public abstract void accept(PrintVisitor v);

    public abstract void accept(ShowVisitor v);
}

class Composite extends Graphic{

   public void accept(Visitor v){
     v.visit(this);
   }

   public void accept(PrintVisitor v)'
     v.visit(this);
   }

   public void accept(showVisitor v){
     v.visit(this);
   }
   ...
}
```

Nous remarquons que la méthode void accept(Visitor v) n'est pas utilisée vu que la partie Composite communique avec la partie Visiteur à partir des méthodes *accept* qui se réfèrent directement aux classes Visiteurs concrètes. Cette méthode est supprimée de de toute la hiérarchie du Composite à l'aide de l'opération *DeleteMethodInHierarchy*.

Étape III. Pour déplacer le code vers la partie Composite, il suffit de faire un *inline* pour les appels des méthodes *visit* dans la partie Composite. Mais nous devons tout d'abord supprimer les déclarations abstraites des méthodes *visit* de la classe *Visitor* (voir code source ci-dessous) car l'outil de refactoring ne permet pas d'effectuer cette tâche de refactoring avec les méthodes abstraites.

```
abstract class Visitor{
  public abstract void visit(Ellipse e);

  public abstract void visit(Composite c);
}
```

La suppression de ces déclarations abstraites de la classe Visitor est légale car après l'application de l'étape I, le programme ne contient plus de référence vers les méthodes déclarées dans cette classe. Cette étape est effectuée par l'application de l'opération *PushDownAll*.

Déplacement du code métier vers la partie Composite.

Maintenant, une fois que nous avons fourni les conditions nécessaires pour faire un *inline* pour les appels des méthodes *visit*, nous appliquons l'opération *InlineMethod* (étape IV). Cette opération va déplacer le code métier des classes Visiteurs vers les méthodes *accept* correspondantes dans la partie Composite et supprimer les méthodes *visit* de ces classes. Voici ci-dessous une partie du code du programme qui permet de voir son état à ce stade :

```
class Ellipse extends Graphic{
    public void accept(PrintVisitor v){
```

```
      System.out.println("Ellipse :" + this);
    }

    public void accept(ShowVisitor v){
      System.out.println("Ellipse corresponding to the object " + this + ".");

    }
}



class Composite extends Graphic{
   public void accept(PrintVisitor v){
      System.out.println("Composite:");
      for (Graphic g : graphics) {
            g.accept(new PrintVisitor());
      }
   }

   public void accept(ShowVisitor v){
     System.out.println("Composite " + this + " composed of:");
     for (Graphic g : graphics) {
        g.accept(new showVisitor());
      }
      System.out.println("(end of composite)");
   }
   ...
}
```

Retouches finales pour l'obtention de la structure Composite.

Étape V. Dans l'étape VI (l'étape suivante), nous devons supprimer les paramètres des méthodes *accept* qui ne sont plus utiles. Mais si nous effectuons cette tâche directement avec les méthodes *accept*, le programme sera erroné vu qu'on aura deux méthodes qui portent le même nom et la même signature dans la même portée. Pour éviter ce problème, nous renommons les méthodes *accept* en donnant à chacune un nom temporaire avec la fonction aux. Le choix du nom dépend du Visiteur qui représente le type de paramètre de la méthode en question. Ainsi notre programme sera comme suit :

```
class Composite extends Graphic{
   public void printAux(PrintVisitor v){
      System.out.println("Composite:");
      for (Graphic g : graphics) {
            g.printAux(new PrintVisitor());
      }
   }

   public void showAux(ShowVisitor v){
     System.out.println("Composite " + this + " composed of:");
     for (Graphic g : graphics) {
        g.showAux(new showVisitor());
      }
      System.out.println("(end of composite)");
   }
```

```
   ...
}
```

Étape VI. Après que nous avons associé aux méthodes *accept* des noms différents, nous pouvons ainsi supprimer les paramètres de ces méthodes et qui ne sont plus utiles. Nous appliquons ainsi l'opération *RemoveParameter* qui va supprimer le paramètre de chaque méthode sur toute la hiérarchie du Composite. Voici une partie de programme qui montre son état à ce stade de la transformation :

```
abstract class Graphic{
   public void print(){
       printAux();
   }
   public abstract void printAux();


   public void show(){
       showAux();
   }
   public abstract void showAux();




class Composite extends Graphic{
   public void printAux(){
      System.out.println("Composite:");
      for (Graphic g : graphics) {
            g.printAux();
      }
   }

  ...
}
```

Étape VII. Pour casser la délégation des méthodes métier vers les méthodes auxiliaires qui sont récursives, nous remplaçons l'appel récursif par un appel vers son délégateur. Ceci est légal vue que les méthodes métiers sont des délégateurs à ces méthodes. Dans notre programme cette délégation existe dans la classe Graphic comme suit :

```
   public void print(){
       printAux();
   }
   ...
   public void show(){
       showAux();
   }
   ...
```

Ceci s'effectue à l'aide de l'opération *ReplaceMethodDuplicate* qui va remplacer les appels récursifs des méthodes auxiliaires par leurs délégateurs comme suit :

```
   public void printAux(){
      System.out.println("Composite:");
      for (Graphic g : graphics) {
            g.print();
      }
   }

  ...
```

Étape VIII. Cette étape vise à récupérer les déclarations abstraites des méthodes métier au niveau de la racine du Composite. Nous devons ainsi faire descendre les définitions de ces méthodes vers les sous classes et laisser leurs déclarations abstraites. Pour réaliser ceci, nous appliquons l'opération *PushDownImplementation*. Cette étape représente aussi une étape préparatoire pour récupérer plus tard les redéfinitions des méthodes métier dans les classes Composites. Voici ci-dessous une partie du programme qui montre son état après l'application de cette étape :

```
abstract class Graphic{
   public abstract void print();

   public abstract void printAux();

   ...
}


class Composite extends Graphic{
   public void printAux(){
      System.out.println("Composite:");
      for (Graphic g : graphics) {
            g.print();
      }
   }

   public abstract void print(){
      printAux();
   }

  ...
}
```

Étape IX. Nous avons déjà mentionné que pour faire un *inline* pour une méthode, elle ne doit pas être abstraite, or les méthodes auxiliaires sont encore abstraites au niveau de la racine du Composite. Dans le but de permettre à l'outil de refactoring d'inliner ces méthodes, nous appliquons cette étape qui est une étape préparatoire pour l'étape suivante. Il s'agit de supprimer les déclarations abstraites des méthodes auxiliaires de la racine du Composite. Pour faire ceci, nous appliquons l'opération *PushDownAll* qui va supprimer les déclarations abstract void printAux() et abstract void showAux() de la classe Graphic. Ceci est légal vue que ces méthodes ne sont utilisées que dans les sous classes.

Étape X. Pour supprimer totalement la relation de délégation entre les méthodes métier et les méthodes auxiliaires et supprimer ces méthodes temporaires, nous appliquons l'opération *InlineMethod* qui va récupérer les redéfinitions des méthodes métiers dans les sous classes et supprimer les méthodes auxiliaires comme suit :

```
class Composite extends Graphic{
   public void print(){
      System.out.println("Composite:");
      for (Graphic g : graphics) {
            g.print();
      }
   }


  ...

}
```

Étape. XI et XII. A ce stade de la transformation, la partie essentielle de la structure Composite est récupérée et il n'y a aucune relation entre la partie Composite et la partie Visiteur. La partie Visiteur ne contient que des classes vides qui ne sont pas utilisées et par suite nous pouvons les supprimer. Mais nous devons supprimer tout d'abord les sous classes (PrintVisitor et ShowVisitor dans notre exemple) puis leur classe mère qui est référencée par ses sous classes avec la relation de l'héritage.

Résultat.

Après l'application de cette transformation nous obtenons un programme qui structuré selon le patron Composite et qui est équivalent au programme de la figure 7. Le programme résultant de cette transformation peut être différent de celui de la figure 7 dans l'ordre d'emplacement des méthodes ou dans la perte de certains commentaires, mais il assure le même comportement que celui-ci. Cette transformation automatique prend à peu près 4 secondes.

6 Bilan

Nous avons présenté dans ce chapitre deux algorithmes de transformation qui permettent le passage automatique et réversible entre deux versions simples des patrons Composite et Visiteur. Ces algorithmes sont composés par des opérations de refactoring. Vu que les opérations de refactoring sont sensés préserver la sémantique par définition, nous considérons que notre transformation préserve aussi la sémantique.

Le tableau suivant donne une idée sur le profil de notre transformation en matière de temps d'exécution et en fonction de la taille du programme et par rapport à son application à partir du menu de refactoring :

Degré d'automatisation	Taille du programme	Temps d'exécution
Automatique (notre transformation)	5 classes	4 s.
Semi-automatique	5 classes	720 s. (12 Min.)

Cette transformation a été appliqué sur une version primitive du patron Composite : méthodes métier sans paramètres, sans types de retour, avec un seul niveau hiérarchique et avec une racine sous forme d'une classe abstraite et non pas une interface.

Les transformations présentées dans ce chapitre seront par la suite vérifiées à l'aide du calcul de la précondition qui garantit leur réussite (chapitre 4). Ensuite, nous étendons ces transformations afin de les appliquer sur d'autres variations des patrons Composite et Visiteur (chapitre 5). Et enfin, nous les appliquons sur un programme réel JHotDraw (chapitre 6).

4

Calcul des Préconditions de la transformation

Sommaire

Dans ce chapitre, nous présentons notre application du calcul des préconditions afin de garantir *statiquement* le bon déroulement de la transformation présentée dans le chapitre précédent. Les préconditions générées garantissent le bon déroulement de cette transformation avant son lancement : toutes les préconditions des opérations composant la transformation seront vérifiées à l'exécution.

Notre contribution consiste à définir toutes les données nécessaires pour alimenter le système de calcul des préconditions choisi. Ces données représentent les préconditions et les rétro-descriptions (*backward descriptions*) de chaque opération de refactoring (le terme *backward decriptions*/rétro-descriptions est expliqué dans le chapitre 2, page 30).

1 Système de calcul des préconditions

Choix du système de calcul

Nous avons vu dans le chapitre 2 deux systèmes de calcul de préconditions. Nous utilisons dans la suite le système de calcul proposé par Kniesel et Koch [KK04]. Ce choix vient du fait que ce système de calcul permet de générer les préconditions d'une séquence d'opérations de refactoring en lui fournissant les rétro-descriptions qui représentent des règles de calcul (voir page 30). Alors que, l'autre système de calcul proposé par O'Cinnéide [OC00] converge plus vers la génération des postconditions d'une transformation pour des préconditions données. Or, notre objectif dans cette thèse est de générer la bonne précondition de la transformation et non pas les postconditions.

Utilisation pratique du système

Le système de calcul proposé par Kniesel et Koch [KK04] fournit uniquement un calcul, nous discutons ici des usages pratiques de calcul.

Non-échec

Le non échec d'une transformation signifie que chaque opération élémentaire inclue dans la transformation s'applique sans échec par la satisfaction de sa précondition tout en prenant en compte le nouveau état du programme après l'application de l'opération qui la précède. Ceci garantit que la précondition de la composition des opérations de refactoring est satisfaite, ce qui permet d'éviter les cas d'échec de la transformation en cours de son exécution.

Correction de l'effet

Pour garantir que la transformation a un effet correct sur le programme à transformer, on doit fixer les propriétés de la structure cible du programme. Ceci se fait par l'identification des bonnes rétro-descriptions. La bonne spécification des rétro-descriptions permet aussi l'inférence des bonnes préconditions lors du calcul de la précondition minimale. Ceci permet d'assurer le départ de la bonne structure initiale du programme et l'obtention de la bonne structure cible qui lui correspond.

Préservation de la sémantique

Pour s'assurer qu'une composition d'un ensemble d'opérations de refactoring préserve la sémantique, il faut montrer que chaque opération de la composition préserve la sémantique. Notre hypothèse dans cette thèse est que toute opération de refactoring préserve la sémantique. Cette hypothèse reste relative, car les opérations de refactoring ne sont pas toutes prouvées correctes.

Lorsque nécessaire, pour appuyer notre hypothèse, nous proposons des préconditions plus fortes que celles de l'outil de refactoring pour vérifier que chaque opération est utilisée dans un contexte où elle préserve la sémantique. Dans ce contexte, nous ne sommes pas obligés de modifier les outils de refactoring mais nous exigeons juste une sorte de restriction sur les préconditions de chaque opération utilisée dans le calcul.

Réversibilité de la transformation

Après l'application d'une transformation aller-retour d'un programme structuré selon le patron Composite vers un programme structuré selon le patron Visiteur, le programme récupéré en tant que programme de départ doit toujours satisfaire les préconditions pour appliquer un autre tour de cette transformation, et ceci peut-être itéré. Cette propriété permet d'une part de garantir qu'ont peut toujours faire des allers et retours à partir et vers le programme initial, et d'autre part de vérifier que le programme en question n'a pas subi une modification qui viole la précondition initiale.

2 Prédicats proposés

Un prédicat dans les outils de refactoring est implémenté sous la forme d'une analyse statique du programme pour vérifier une propriété dans ce programme. Dans notre thèse, nous proposons des prédicats pour vérifier nos transformations. Ces prédicats ne sont pas encore implémentés par une analyse statique.

Parmi les prédicats que nous proposons, certains sont déjà définis dans l'approche de Kniesel et Koch [KK04] (décrite dans le chapitre 2, page 4) comme par exemple $ExistsMethod$ ou $ExistsClass$. En effet, nous nous contentons ici de les réutiliser tels qu'ils sont. D'autres prédicats sont utilisés dans leur approche, mais nous ne les utilisons pas, comme par exemple $CanModifyClass$ car nous les considérons qu'ils sont vérifiés par défaut (il s'agit dans cet exemple le droit d'accès aux fichiers du code source).

Le choix de définir nos propres spécifications et non pas celles des travaux précédents, vient aussi du fait que pour une même opération qui existait dans ces travaux, on peut avoir plusieurs contextes d'utilisations. En effet, au contraire de ces travaux, nous préférons par exemple générer différentes versions d'une même opération pour différents contextes au lieu d'utiliser une seule opération dont sa spécification sera plus complexe. Ceci donne à notre spécification plus d'explicité et plus de souplesse vis-à-vis aux contextes d'utilisation envisageables.

Prédicats sur les méthodes

ExistsMethod$(classname, methodname)$

La méthode $methodname$ existe dans la classe $classname$. L'existence de la méthode peut se manifester selon qu'elle est définie dans la classe ou qu'elle est déclarée abstraite, mais, on ne considère pas ici l'existence par héritage. La méthode $methodname$ peut être surchargée.

ExistsMethodWithParamTypes$(classname, methodname, parameterList)$

La méthode $methodname(parametyerList)$ existe dans la classe $classname$. La méthode en question peut être définie dans la classe ou abstraite. La méthode $methodname$ peut être surchargée.

IsInheritedMethod$(classname, methodname)$

La méthode $methodname$ est héritée par la classe $classname$ sans être redéfinie dans cette classe. Ce prédicat lorsque il est utilisé avec négation veut dire que toute méthode qui porte le nom $methodname$ n'est pas héritée par la classe $classname$ car il ne porte aucune information sur la signature de la méthode en question.

IsInheritedMethodWithParams$(classname, methodname, parameterList)$

La méthode $methodname(paramList)$ est héritée par la classe $classname$. La méthode peut être redéfinie ou non.

Remarque. Les prédicats ExistsMethod et IsInheritedMethod ne montrent pas la signature de la méthode à vérifier. Ceci vient dans le contexte où on veut par exemple vérifier que toute méthode avec le nom de la méthode en question n'existe pas ou n'est pas héritée indifféremment de sa signature. Il s'agit des prédicats qui peuvent être utilisées pour exprimer des préconditions fortes.

ExistsParameterWithType$(classname, methodname, parameterList, type)$

Le type $type$ existe dans la liste des types des paramètres de la méthode $methodname$ de la classe $classname$. Ce prédicat ne nécessite pas une analyse statique du programme car on peut le vérifier directement à partir de ses arguments $type$ et $parameterList$, mais il représente d'une manière plus expressive le prédicat Contains$(parameterList, type)$

ExistsParameterWithName$(classname, methodname, parameterList, parameter)$

Le paramètre de nom $parameter$ existe dans la liste des paramètres de la méthode $methodname$ de la classe $classname$. La même remarque sur le prédicat précédent est applicable ici pour $parameter$ et $parameterList$.

ExistsMethodInvocation$(classname, callermethod, definingclass, calledmethod)$

La méthode $calledmethod$ de la classe $definingclass$ est invoquée dans le corps de la méthode $callermethod$ définie dans la classe $classname$. Voici ci dessous un exemple du programme présenté dans le chapitre 3 :

```
class  Composite{
   ...
   public void accept(PrintVisitor v){
      v.visit(this);
   }
   ...
}

class PrintVisitor{
   ...
   public void visit(Composite c){...}
   ...
}
```

Dans cet exemple ExistsMethodInvocation$(Composite, accept, PrintVisitor, visit)$ indique que la méthode visit(Composite) qui est définie dans la classe PrintVisitor est invoquée dans le corps de la méthode accept qui est définie dans la classe Composite. L'objet v qui invoque la méthode visit ne doit pas avoir un type dynamique différent de PrintVisitor (éviter le changement de la sémantique lorsque la méthode visit est redéfinie dans la classe qui correspond à ce type dynamique).

ExistsAbstractMethod$(classname, methodname)$

La méthode $methodname$ est abstraite dans la classe $classname$ et n'est pas héritée.

IsDelegator$(classname, methodname, deleguee)$

La méthode $methodname$ de la classe $classname$ est un délégateur vers la méthode $deleguee$. Par exemple la méthode print de la classe Graphic est un délégateur de la méthode printAux dans le code source suivant :

```
abstract class Graphic{
   ...
   public void print(){
       printAux();
   }
   public abstract void printAux();
   ...
}
```

IsInverter$(clasname, methodname, invertedtype, returntype)$

La méthode $methodname$ de la classe $classname$ est un délégateur vers une autre méthode invoquée dans sa portée par un objet de type $invertedtype$ et qui a un type de retour $returntype$. Voici ci-dessous un exemple qui donne une idée sur ce prédicat :

```
class Composite{
   ...
   public void accept(Visitor v){
      v.visit(this);
```

```
    }
    ...
}
```

Dans ce code source, IsInverter$(Composite, accept, Visitor, void)$ indique que la méthode accept de la classe Composite qui est un délégateur de la méthode visit qui est invoquée par l'objet v qui est de type Visitor et dont le type de retour dans cet exemple est void.

Lorsque deux méthodes vérifient ce prédicat vers la même méthode, elles sont considérées égales du point de vue sémantique. Voici ci dessous un exemple qui explique cette remarque :

```
class Composite{
    ...
    public void printAux(Visitor v){
       v.visit(this);
    }
    public void showAux(Visitor v){
       v.visit(this);
    }
}

abstract class Visitor{
    abstract void visit(Composite c);
}
```

Les deux méthodes printAux et showAux sont considérées égales de point de vue sémantique, car elles sont délégateurs de la même méthode visit qui est invoquée dans les deux cas par un objet v du type Visitor.

IsOverloaded$(clasname, methodname)$

La méthode $methodname$ de la classe $classname$ est surchargée.

IsIndirectlyRecursive$(classname, methodname)$

la méthode $methodname$ de la classe $classname$ est indirectement récursive. La méthode $methodname$ appelle une méthode qui elle même contient un appel vers elle. Le code source suivant clarifie l'utilisation de ce prédicat :

```
abstract class Graphic{
    public void print(){
        printAux();
    }
    public abstract void printAux();

    ...
}

class Composite extends Graphic{
   ...
   public void printAux(){
       System.out.println("Composite:");
       for (Graphic graphic : graphics) {
             graphic.print();
       }
    }
   ...
```

```
}
```

Nous remarquons que la méthode printAux de la classe Composite est indirectement récursive par l'intermédiaire de la méthode print qui est définie dans la classe Graphic et non pas dans la classe Composite.

IsVisible$(classname, elementname, anotherclass)$

La méthode/varaiable $elementname$ de la classe $classname$ est visible par la classe $anotherclass$. L'argument $elementname$ peut être une méthode ou une variable d'instance.

IsOverriding(c, m)

La méthode m dans la classe c est une redéfinition de celle héritée de sa classe mère. La méthode m peut être surchargée.

IsOverridden(c, m)

La méthode m définie dans la classe c est redéfinie au moins dans l'une de ses sous-classes. La méthode m peut être abstraite.

HasReturnType$(classname, methodname, returntype)$

La méthode $methodname$ de classe $classname$ a le type de retour $returntype$. La méthode m peut être surchargée.

MethodIsUsedWithType$(classname, methodname, definedparametertype, usedparametertype)$

La méthode $methodname(definedparametertype)$ de la classe $classname$ est utilisée avec les types $usedparametertype$ qui est un sous type de $definedparametertype$. Le code source suivant clarifie l'utilisation de ce prédicat :

```
abstract class Graphic{
   ...
   public void print(){
       printAux(new PrintVisitor());
   }
   public abstract void printAux(Visitor v);
   ...
}
```

Le méthode printAux de la classe Graphic qui a un paramètre v de type Visitor est utilisée dans la portée de la méthode print avec le paramètre de type PrintVisitor (un sous type de Visitor).

AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded$(classname, methodname, object)$

Les méthodes qui sont invoquées dans le corps de $methodname$ avec $object$ comme argument ne sont pas surchargées. Ce prédicat est utile pour éviter le changement de la liaison statique lorsque $object$ change de type. Le code source suivant clarifie l'utilisation de ce prédicat :

```
class PrintVisitor{
   void visit(Composite c){
     ...
   }
```

```
}

class Composite{
   ...
   public void printAux(PrintVisitor v){
       v.visit(this);
   }
   ...
}
```

La méthode visit qui est définie dans la classe PrintVisitor et qui est invoquée dans la portée de la méthode Composite : :printAux avec this comme argument n'est pas surchargée. Ceci est vérifié par exemple si on veut faire descendre la méthode Composite : :printAux vers sa sous classe, ce qui peut changer la liaison statique si la méthode visit est surchargée.

AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC$(classname, methodname, object, type)$)

Tous les appels des méthodes invoquées par l'objet $object$ dans le corps de la méthode $methodname$ sont légaux si on change le type de $object$ et si la méthode appelée est déclarée abstraite dans le nouveau type qui est le super type du premier. Le code source suivant clarifie l'utilisation de ce prédicat :

```
abstract class Visitor{
   abstract void visit(Composite c);

}

class PrintVisitor extends Visitor{
   void visit(Composite c){
     ...
   }

}

class Composite{
   ...
   public void printAux(PrintVisitor v){
       v.visit(this);
   }
   ...
}
```

Lorsque on change le type de paramètre v de la méthode printAux vers $Visitor$ (la classe mère de $PrintVisitor$), on est sûr que la méthode visit qui est invoquée par cet objet est déclarée abstraite dans cette classe.

IsUsedMethod$(classname, methodname, types)$

La méthode $methodname(types)$ de la classe $classname$ est utilisée dans cette classe ou ailleurs. Nous nous intéressons uniquement à savoir si cette méthode est utilisée ou non peu n'importe la portée dans laquelle est utilisée.

IsUsedMethodIn$(classname, methodname, callerclass)$

Ce prédicat est une restriction du prédicat IsUsedMethod : la méthode $methodname$ de la classe $classname$ est utilisée dans la portée de la classe $callerclasse$.

IsRecursive($clasname, methodname$)

La méthode $methodname$ de la classe $clasname$ est récursive. Par exemple la méthode print dans le code source suivant est récursive :

```
abstract class Graphic{
  abstract void print();
  ...
}

class Composite extends Graphic{
   ...
   public void print(){
      System.out.println("Composite:");
      for (Graphic graphic : graphics) {
            graphic.print();
      }
   }
   ...
}
```

Prédicats sur les classes

ExistsClass($classname$)

La classe $classname$ existe. Nous ne considérons pas ici la vérification sur les paquetages, mais la vérification porte sur un paquetage donnée.

ExtendsDirectly($classname, superclass$)

La classe $classname$ est une sous-classe directe de la classe $superclass$. Il n'existe pas de classes intermédiaires entre $classname$ et $superclas$.

IsAbstractClass($classname$)

La classe $classname$ est abstraite.

AllSubclasses($superclass, [classname1, classname2]$)

Les classes $classname1$ et $classname2$ sont des sous-classes de la classe $superclass$. Ce prédicat est intéressant pour notre transformation vu que dans certaines étapes de la transformation nous nous intéressons à parcourir toute la hiérarchie de classes. Ceci permet d'empêcher l'utilisateur de mettre des classes qui n'appartiennent pas à la hiérarchie.

Prédicats sur les constructeurs

IsUsedConstructorAsMethodParameter($classname, classofmethodname, methodname$)

Le constructeur de la classe $classname$ est utilisé comme un paramètre de la méthode $methodname$ définie dans la classe $classofmethodname$. Voici ci-dessous un exemple qui clarifie le rôle de ce prédicat :

```
abstract class Graphic{
   ...
   public void print(){
       printAux(new PrintVisitor());
```

```
    }
    ...
}
```

Dans cet exemple le constructeur de la classe PrintVisitor est utilisé comme un paramètre pour la méthode printAux.

IsUsedConstructorAsInitializer($classname, classofinitializingmethod, initializingmethod$)

Le constructeur de la classe $classname$ est utilisé pour créer un nouvel objet dans la portée de la méthode $initializingmethod$ définie dans la classe $classofinitializingmethod$. Ce prédicat est utilisé surtout lorsque on veut supprimer $classname$ afin de vérifier que cette classe n'est pas référencée dans le programme.

IsUsedConstructorAsObjectReceiver($classname, classofinvokingmethod, invokingmethod$)

Le constructeur de la classe $classname$ est utilisée comme un objet qui invoque une méthode dans la portée de la méthode $invokingmethod$ définie dans la classe $classofinvokingmethod$. Voici ci-dessous un exemple clarifiant le rôle de ce prédicat :

```
class Composite{
    ...
    public void accept(PrintVisitor v){
        v.visit(this);
    }
    ...
}
```

Dans cet exemple nous pouvons dire que le constructeur de la classe PrintVisitor est utilisé comme un objet receveur dans la portée de la méthode accept. Ce prédicat supporte aussi les objets receveurs qui présentent des instanciations du constructeur comme new PrintVisitor().visit(this).

Prédicats sur les types

ExistsType($type$)

Le type $type$ existe.

IsGenericsSubtype($classname, TypeParameter1, superclassname, TypeParameter2$)

La classe $classname$ qui est paramétrique selon le type $TypeParameter1$ est une sous classe de la classe $superclassname$ qui est paramétrique selon le type $TypeParameter2$. Nous allons utiliser ce prédicat dans le chapitre 5 lorsque nous allons traiter les variations des patrons Composite et Visiteur. L'exemple suivant montre une hiérarchie de types génériques :

```
class Visitor <T> {...}

class PerimeterVisitor extends Visitor <Integer>

class ToStringVisitor extends Visitor <String>
```

Dans cet exemple IsGenericsSubtype($PerimeterVisitor, Integer, Visitor, T$) est vraie.

IsSubtype($subtype, type$)

Le type $subtype$ est un sous-type du type $type$.

IsPrimitiveType($typename$)

Le type $typename$ est un type primitif. Par exemple int est un type primitif, par contre le type $Integer$ n'est pas primitif.

Prédicats sur les attributs et les variables

ExistsField($classname, fieldname$)

Le champs $fieldname$ existe dans la classe $clasname$.

ExistsFieldInMethodScope($classname, methodname, fieldname$)

Le champs $fieldname$ existe dans la portée de la méthode $methodname$ définie dans la classe $classname$.

IsInheritedField($classname, fieldname$)

Le champs $fielname$ est hérité par la classe $classname$.

IsUsedField($classname, attributename, methodname$)

Le champs $attributename$ de la classe $classname$ est utilisée dans la méthode $methodname$ de la même classe. Par exemple dans le code source suivant ce prédicat est vraie pour la collection graphics déclarée dans la classe Composite et utilisée dans la portée de la méthode print :

Dans la classe $Composite$:

```
class compositeGraphic{
   ...
   ArrayList<Graphic>  graphics = new ArrayList<Graphic>;

   public void print(){
      System.out.println("Composite:");
      for (Graphic graphic : graphics) {
            graphic.print();
      }
   }
   ...
}
```

Ce prédicat est utilisé généralement lorsque on veut déplacer une méthode pour vérifier si les champs de sa classe d'origine sont utilisés ou non dans la portée de cette méthode car ceci va influencer les références vers ces champs dans la classe cible.

BoundVariableInMethodBody($classname, methodname, variablename$)

La variable $variablename$ est utilisée dans le corps de la méthode $methodname$ définie dans la classe $classname$.

Prédicats sur la visibilité

IsPublic($classname, elementname$)

$elementname$ déclaré dans la classe $classname$ est publique. Le paramètre $elementname$ du prédicat peut être une méthode ou un champs.

IsPrivate($classname, elementname$)

$elementname$ déclaré dans la classe $classname$ est privé.

IsProtected($classname, elementname$)

$elementname$ déclaré dans la classe $classname$ est protégé.

3 Formules logiques et rétro-descriptions

Nous présentons ici les notations que nous utilisons pour exprimer les préconditions et les rétro-descriptions de chaque opération de refactoring. Dans le chapitre 2 (page 30) nous avons déjà présenté les notations utilisées par Kniesel et Koch [KK04]. Dans cette thèse nous allons utiliser les mêmes mais avec quelques différences qui seront mentionnés dans cette section.

Formules

Pour décrire les préconditions d'une opération de refactoring, nous utilisons des formules logiques. Une formule logique est de la forme suivante :

$$\boxed{F ::= P(A, B...)|F|\neg F|F \wedge F|F \vee F|\top|\bot}$$

Avec :

- F : formule.
- P : proposition avec une liste d'arguments (celles définies dans la section 2).
- $\neg F$: négation d'une formule.
- $F \vee F$: disjonction entre deux formules.
- $F \wedge F$: conjonction de deux formules,
- $\top$: vrai
- $\bot$: faux

Notations pour les rétro-descriptions

Pour toute opération de base Op, les rétro-descriptions de Op sont décrites par un ensemble de règles de calcul de la forme suivante :

$$\boxed{P \mapsto Q, \text{ avec } P \text{ et } Q \text{ sont des propositions}}$$

Par exemple, parmi les rétro-descriptions de l'opération *RenameClass(A,B)* nous citons :

- $\mathsf{ExistsMethod}(B, m) \mapsto \mathsf{ExistsMethod}(A, m)$: la méthode m sera définie dans la classe B, si avant elle est définie dans la classe A.
- $\mathsf{ExistsClass}(A) \mapsto \bot$: $\mathsf{ExistsClass}(A)$ est vrai après si $\bot$ est vrai avant. Ceci revient à dire que $\mathsf{ExistsClass}(A)$ est toujours faux après.
- $\mathsf{ExistsClass}(B) \mapsto \top$: $\mathsf{ExistsClass}(B)$ est vrai après si $\top$ est vrai avant. Ceci revient à dire que $\mathsf{ExistsClass}(B)$ est toujours vrai après.

Remarques.

Variables pour des valeurs non précisées. Dans les rétro-descriptions nous pouvons utiliser des variables qui peuvent être instanciées par n'importe quelle valeur (le nom de la variable s'écrit en majuscule). Par exemple, on utilise V qui peut être instanciée par n'importe quel nom de méthode qui existe dans la classe A et qui ne sera plus existante après le renommage de cette classe et on écrit : $\mathsf{ExistsMethod}(A, V) \mapsto \bot$.

Propriétés inchangées. Nous n'explicitons pas les propriétés qui restent inchangées après l'application d'une opération de refactoring dans la rétro-descriptions (dans Kniesel et Koch [KK04], on les appelle *self*). Par exemple, lorsqu'on applique l'opération *RenameClass(A,B)*, les types de retour des méthodes de la classe A ne changent pas.

Sémantique des rétro-descriptions. Selon le sens qu'on lui donne, la précondition exprimée par une rétro-description pourrait exprimer une condition nécessaire, sufisante ou nécessaire et suffisante. Examinons ces possibilités avec un exemple.

On considère l'opération théorique *moveAllMethods(A,B)* qui est sensée déplacer toutes le méthodes de la classe A vers la classe B. Considérons à présent la rétro-description suivante pour cette opération :

$$\mathsf{ExistsMethodDefinition}(B, m) \mapsto \mathsf{ExistsMethodDefinition}(A, m)$$

Selon la sémantique choisie pour les rétro-descriptions, celle-ci pourrait être interprétée des manières suivantes :

- Pour qu'il y ait une méthode m dans la classe B après l'opération, *il suffit* qu'il y ait une méthode m dans la classe A.
- Pour qu'il y ait une méthode m dans la classe B après l'opération, *il faut et il suffit* qu'il y ait une méthode m dans la classe A.

Si on considère la première sémantique, la règle ci-dessus décrit bien l'opération. En revanche, cette règle ne s'applique pas dans un contexte négatif : on ne peut pas déduire que pour qu'il n'y ait pas de méthode m dans B il suffit qu'il n'y ait pas de méthode m dans A.

Dans le cas d'une sémantique exprimant une condition nécessaire et suffisante, la règle est fausse (il n'est pas *nécessaire* d'avoir une méthode m dans A pour trouver une méthode m dans B après l'opération). La règle suivante est correcte en revanche :

$$\mathsf{ExistsMethodDefinition}(B, m) \mapsto (\mathsf{ExistsMethodDefinition}(A, m) \vee \mathsf{ExistsMethodDefinition}(B, m))$$

De plus, ici on peut appliquer la règle dans un contexte négatif : pour qu'il n'y ait pas de méthode m dans B, il faut et il suffit qu'il n'y ait pas de méthode m dans A et pas de méthode m dans B.

Dans le calcul que nous utilisons, **nous adoptons la sémantique de précondition nécessaire et suffisante**. Les règles de rétro-descriptions données dans tout le document et le calcul de composition statique se conforment à cette sémantique.

Les rétro-descriptions et les préconditions décrivant les opérations de refactoring utilisées sont données dans la section suivante et en annexe.

4 Description des opérations de refactoring de la transformation

Les opérations constituant nos algorithmes de transformation sont de deux catégories :

- Des opérations élémentaires comme l'opération *CreateIndirectionInSuperClass* (section 2) qui se base sur l'opération *change signature* de l'outil de refactoring de IntelliJ IDEA.
- Des opérations composées d'une séquence d'autres opérations comme l'opération *MergeDuplicateMethods* (section 4).

Nous détaillons ces deux opérations en présentant leurs préconditions et rétro-descriptions. Pour les autres opérations de refactoring, elles sont détaillées dans l'annexe B.

Opération élémentaire : *CreateIndirectionInSuperClass*

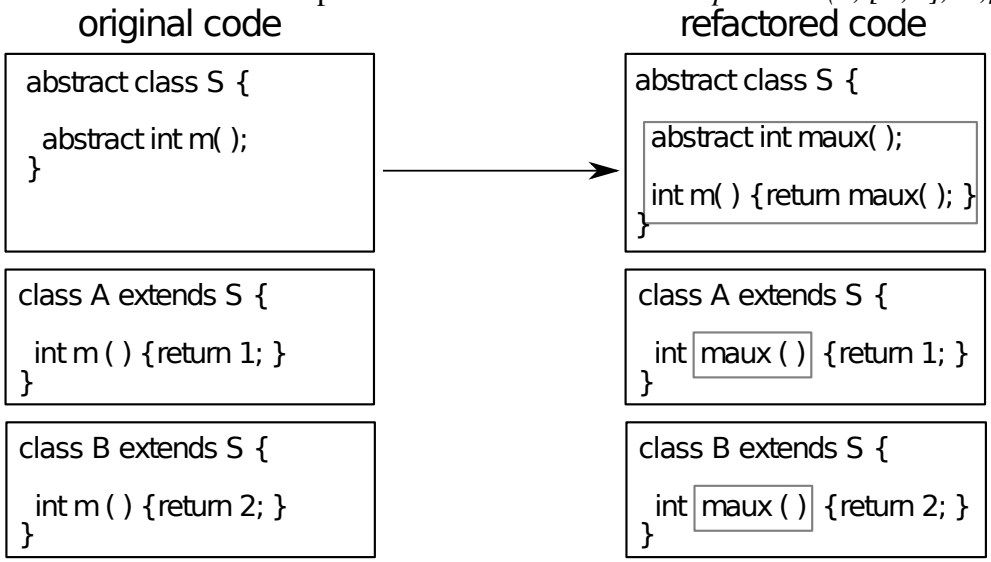
Cette opération est appliquée dans l'étape 2 de l'algorithme 3 de transformation du Composite vers le Visiteur (page 33).

Rôle

CreateIndirectionInSuperclass (s, [a,b], m,[t,t'], r,n) : cette opération est utilisée pour créer un déléguée de la méthode $s :: m$ qui a les paramètres t,t',... et avec le type de retour r et l'appeler n. Les sous classes $a, b, ...$ de s seront influencées par ce changement. Voici ci dessous un exemple de la création d'un déléguée n() d'une méthode int s : :m(). Cette opération est utilisée sur une méthode qui est déclarée dans une classe abstraite et définie sur ses sous classes. Nous supposons aussi que cette opération s'applique sur les méthodes non surchargées.

Noter que les arguments de l'opération *CreateIndirectionInSuperclass* présentés ici sont des méta variables qui seront substituées par des constantes lorsque on applique le calcul de la précondition sur un programme.

Illustration de l'utilisation de l'opération *CreateIndirectionInSuperclass (S, [A,B], m,[],int,n)*



Outil de refactoring : IntelliJ IDEA

Cette opération se base sur l'opération *change signature* de l'outil de refactoring IntelliJ IDEA. Elle est utilisée à partir de l'outil de refactoring comme suit :

1. Mettre le curseur sur la méthode $s :: m$.
2. Cliquer sur *change signature* du menu de refactoring.
3. Choisir le nom du déléguée, par exemple n.
4. Choisir l'option *Delegate via overloading method* : ceci permet de transformer la méthode methodname en un délégateur de la nouvelle méthode newname.

En pratique, et pour des raisons d'automatisation de la transformation nous utilisons l'appel des opérations à partir de l'*API* et non pas à partir de *GUI*.

Préconditions

Voici ci dessous la liste des préconditions [1] que nous avons construit à partir des prédicats présentés dans la section 2.

$$
\begin{aligned}
&\texttt{ExistsClass}(s)\\
&\wedge \texttt{ExistsClass}(a)\\
&\wedge \texttt{ExistsClass}(b)\\
&\wedge \texttt{AllSubclasses}(s, [a; b])\\
&\wedge \texttt{IsAbstractClass}(s)\\
&\wedge \texttt{ExistsMethodWithParams}(s, m, [t; t'])\\
&\wedge \texttt{ExistsAbstractMethod}(s, m, [t; t'])\\
&\wedge \texttt{ExistsMethod}(s, m)\\
&\wedge \texttt{ExistsMethod}(a, m)\\
&\wedge \texttt{ExistsMethod}(b, m)\\
&\wedge \neg\texttt{IsInheritedMethod}(s, n)\\
&\wedge \neg\texttt{IsInheritedMethodWithParams}(s, n, [t; t'])\\
&\wedge \neg\texttt{ExistsMethodWithParams}(s, n, [t; t'])\\
&\wedge \texttt{HasReturnType}(s, m, r)\\
&\wedge \texttt{HasReturnType}(a, m, r)\\
&\wedge \texttt{HasReturnType}(b, m, r)\\
&\wedge \neg\texttt{IsPrivate}(s, m)\\
&\wedge \neg\texttt{IsPrivate}(a, m)\\
&\wedge \neg\texttt{IsPrivate}(b, m)\\
&\wedge \neg\texttt{ExistsMethod}(s, n)\\
&\wedge \neg\texttt{ExistsMethod}(a, n)\\
&\wedge \neg\texttt{ExistsMethod}(b, n)
\end{aligned}
$$

Ces préconditions servent à vérifier les propriétés suivantes :

Existence d'une hiérarchie de classes. L'opération est restreinte à une hiérarchie de classes comme c'est indiqué dans sa définition. Cette propriété est exprimée par la formule suivante :

$$
\begin{aligned}
&\texttt{ExistsClass}(s)\\
&\wedge \texttt{ExistsClass}(a)\\
&\wedge \texttt{ExistsClass}(b)\\
&\wedge \texttt{AllSubclasses}(s, [a; b])
\end{aligned}
$$

[1] Ces préconditions ne sont pas encore comparées à celles implémentés par IntelliJ IDEA pour cette opération. Une étude est prévue dans ce contexte.

Changement de la méthode m sur toute la hiérarchie de classes. La méthode s : :m doit être abstraite pour que les méthodes qui l'implémentent dans les sous classes de s se changent en n. L'opération *change signature* qui constitue cette opération peut être aussi utilisée avec des méthodes non abstraites mais dans ce cas d'utilisation, seule la méthode en question change, mais l'utilisation de cette opération dans notre transformation vise à appliquer les changements sur toute la hiérarchie de classes. Cette propriété est vérifiée par la précondition suivante :

$\texttt{IsAbstractClass}(s)$
$\wedge\, \texttt{ExistsAbstractMethod}(s, m)$

La précondition $\texttt{IsAbstractClass}(s)$ est obligatoire dans ce contexte vu qu'une méthode abstraite ne peut être déclarée que dans une classe abstraite.

La méthode m est définie dans toutes les sous classes de la racine de la hiérarchie de classes. Cette propriété est vérifiée par la précondition suivante :

$\texttt{ExistsMethod}(s, m)$
$\wedge\, \texttt{ExistsMethod}(a, m)$
$\wedge\, \texttt{ExistsMethod}(b, m)$

Il s'agit d'une précondition forte qui exige que la méthode s : :m soit définie dans toutes les sous classes de la classe S. Cette opération ne tolère pas les définitions aléatoires de la méthode m. Cette restriction permet à cette opération de s'appliquer uniquement dans ce contexte.

Assurer les bonnes conditions pour créer la méthode n. On doit vérifier que la méthode avec le nom n ne figure pas dans le programme (soit par définition dans une classe, soit par héritage). Ceci est vérifié par la précondition suivante :

$\neg\texttt{IsInheritedMethod}(s, n)$
$\wedge\, \neg\texttt{ExistsMethodWithParams}(s, n, [t; t'])$
$\wedge\, \neg\texttt{IsInheritedMethodWithParams}(s, n, [t; t'])$
$\wedge\, \neg\texttt{ExistsMethod}(s, n)$
$\wedge\, \neg\texttt{ExistsMethod}(a, n)$
$\wedge\, \neg\texttt{ExistsMethod}(b, n))$

Préconditions optionnelles. Nous avons enrichi quelques opérations par des préconditions qui ne sont pas nécessaires pour l'opération elle même mais qui facilitent le calcul des préconditions de la totalité de la transformation en simplifiant les préconditions d'autres opérations qui s'exécutent avant ou après l'opération. Parmi ces préconditions, nous trouvons :

$\neg\texttt{IsPrivate}(s, m)$
$\wedge\, \neg\texttt{IsPrivate}(a, m)$
$\wedge\, \neg\texttt{IsPrivate}(b, m))$

Cette précondition vérifie que la méthode m ne doit pas être privée sur toute la hiérarchie. Ceci est vraie car une méthode qui est déclarée abstraite dans une classe et définie dans ses sous classes ne doit pas être privée. Mais cette précondition est optionnelle ici car le fait qu'on vérifie que la méthode m est déclarée abstraite dans s et définie dans a et b, on peut déduire que la méthode m n'est pas privée.

Rétro-description

Voici ci-dessous la rétro-description de l'opération *CreateIndirectionInSuperclass* que nous avons spécifié et qui décrit l'état du programme après l'application de cette opération :

$\texttt{ExistsAbstractMethod}(s, n) \mapsto \texttt{ExistsAbstractMethod}(s, m)$
$\texttt{ExistsAbstractMethod}(s, m) \mapsto \bot$
$\texttt{IsDelegator}(s, m, n) \mapsto \top$
$\texttt{IsDelegator}(a, n, V) \mapsto \texttt{IsDelegator}(a, m, V)$

$\texttt{IsDelegator}(b, n, V) \mapsto \texttt{IsDelegator}(b, m, V)$
$\texttt{IsDelegator}(a, V, n) \mapsto \texttt{IsDelegator}(a, V, m)$
$\texttt{IsDelegator}(b, V, n) \mapsto \texttt{IsDelegator}(b, V, m)$
$\texttt{IsInheritedMethodWithParams}(s, n, [t; t']) \mapsto \bot$
$\texttt{IsOverriding}(s, n) \mapsto \bot$
$\texttt{ExistsType}(r) \mapsto \top$
$\texttt{HasReturnType}(s, n, [t; t'], r) \mapsto \texttt{HasReturnType}(s, m, r)$
$\texttt{HasReturnType}(a, n, r) \mapsto \texttt{HasReturnType}(s, m, r)$
$\texttt{HasReturnType}(b, n, r) \mapsto \texttt{HasReturnType}(s, m, r)$
$\texttt{ExistsMethodDefinition}(s, n) \mapsto \top$
$\texttt{ExistsMethodDefinition}(a, n) \mapsto \top$
$\texttt{ExistsMethodDefinition}(b, n) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(s, n, [t; t']) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(a, n, [t; t']) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(b, n, [t; t']) \mapsto \top$
$\texttt{ExistsParameterWithName}(s, n, [t; t'], N) \mapsto \bot$
$\texttt{ExistsParameterWithName}(a, n, [t; t'], N) \mapsto \bot$
$\texttt{ExistsParameterWithName}(b, n, [t; t'], N) \mapsto \bot$
$\texttt{ExistsParameterWithName}(s, n, [V], N) \mapsto \texttt{ExistsParameterWithName}(s, m, [V], N)$
$\texttt{ExistsParameterWithName}(a, n, [V], N) \mapsto \texttt{ExistsParameterWithName}(a, m, [V], N)$
$\texttt{ExistsParameterWithName}(b, n, [V], N) \mapsto \texttt{ExistsParameterWithName}(b, m, [V], N)$
$\texttt{ExistsMethodDefinition}(a, m) \mapsto \bot$
$\texttt{ExistsMethodDefinition}(b, m) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(a, m, [t; t']) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(b, m, [t; t']) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(a, m, [V]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(b, m, [V]) \mapsto \bot$
$\texttt{ExistsMethodInvocation}(a, n, s, m) \mapsto \texttt{IsRecursiveMethod}(a, m)$
$\texttt{ExistsMethodInvocation}(b, n, s, m) \mapsto \texttt{IsRecursiveMethod}(b, m)$
$\texttt{ExistsMethodInvocation}(s, m, a, n) \mapsto \top$
$\texttt{ExistsMethodInvocation}(s, m, b, n) \mapsto \top$
$\texttt{BoundVariableInMethodBody}(s, n, V) \mapsto \texttt{BoundVariableInMethodBody}(s, m, V)$
$\texttt{BoundVariableInMethodBody}(a, n, V) \mapsto \texttt{BoundVariableInMethodBody}(a, m, V)$
$\texttt{BoundVariableInMethodBody}(b, n, V) \mapsto \texttt{BoundVariableInMethodBody}(b, m, V)$
$\texttt{IsOverloaded}(s, n) \mapsto \bot$
$\texttt{IsOverloaded}(a, n) \mapsto \bot$
$\texttt{IsOverloaded}(b, n) \mapsto \bot$
$\texttt{IsOverridden}(s, n) \mapsto \bot$
$\texttt{IsOverridden}(a, n) \mapsto \texttt{IsOverridden}(a, m)$
$\texttt{IsOverridden}(b, n) \mapsto \texttt{IsOverridden}(b, m)$
$\texttt{IsOverriding}(a, n) \mapsto \texttt{IsOverriding}(a, m)$
$\texttt{IsOverriding}(b, n) \mapsto \texttt{IsOverriding}(b, m)$
$\texttt{IsRecursiveMethod}(s, n) \mapsto \bot$
$\texttt{IsRecursiveMethod}(a, n) \mapsto \bot$
$\texttt{IsRecursiveMethod}(b, n) \mapsto \bot$
$\texttt{IsIndirectlyRecursive}(a, n) \mapsto \texttt{IsIndirectlyRecursive}(a, m)$

$\texttt{IsIndirectlyRecursive}(b, n) \mapsto \texttt{IsIndirectlyRecursive}(b, m)$
$\texttt{IsIndirectlyRecursive}(a, n) \mapsto \texttt{IsRecursiveMethod}(a, m)$
$\texttt{IsIndirectlyRecursive}(b, n) \mapsto \texttt{IsRecursiveMethod}(b, m)$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(s, n, N, V)$
$\mapsto \texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(s, m, N, V)$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(a, n, N, V)$
$\mapsto \texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(a, m, N, V)$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(b, n, N, V)$
$\mapsto \texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(b, m, N, V)$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(s, n, N)$
$\mapsto \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(s, m, N)$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(a, n, N)$
$\mapsto \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(a, m, N)$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(b, n, N)$
$\mapsto \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(b, m, N)$
$\texttt{IsPrivate}(s, V) \mapsto \bot$
$\texttt{IsPrivate}(a, V) \mapsto \bot$
$\texttt{IsPrivate}(b, V) \mapsto \bot$
$\texttt{IsPrivate}(s, n) \mapsto \bot$
$\texttt{IsPrivate}(a, n) \mapsto \bot$
$\texttt{IsPrivate}(b, n) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(a, n, [t; t']) \mapsto \texttt{IsVisibleMethod}(s, m, [t; t'], a)$
$\texttt{IsInheritedMethodWithParams}(b, n, [t; t']) \mapsto \texttt{IsVisibleMethod}(s, m, [t; t'], b)$
$\texttt{IsVisibleMethod}(s, m, [t; t'], a) \mapsto \top$
$\texttt{IsVisibleMethod}(s, m, [t; t'], b) \mapsto \top$
$\texttt{MethodIsUsedWithType}(a, n, [t; t'], [t; t']) \mapsto \texttt{MethodIsUsedWithType}(a, m, [t; t'], [t; t'])$
$\texttt{MethodIsUsedWithType}(b, n, [t; t'], [t; t']) \mapsto \texttt{MethodIsUsedWithType}(b, m, [t; t'], [t; t'])$
$\texttt{IsUsedMethod}(a, n, [t; t']) \mapsto \texttt{IsUsedMethod}(a, m, [t; t'])$
$\texttt{IsUsedMethod}(b, n, [t; t']) \mapsto \texttt{IsUsedMethod}(b, m, [t; t'])$
$\texttt{IsUsedMethodIn}(a, n, V) \mapsto \texttt{IsUsedMethodIn}(a, m, V)$
$\texttt{IsUsedMethodIn}(b, n, V) \mapsto \texttt{IsUsedMethodIn}(b, m, V)$
$\texttt{IsInverter}(a, n, V, V1) \mapsto \texttt{IsInverter}(a, m, V, V1)$
$\texttt{IsInverter}(b, n, V, V1) \mapsto \texttt{IsInverter}(b, m, V, V1)$
$\texttt{ExistsMethodInvocation}(a, V, V1, n) \mapsto \texttt{ExistsMethodInvocation}(a, V, V1, m)$
$\texttt{ExistsMethodInvocation}(b, V, V1, n) \mapsto \texttt{ExistsMethodInvocation}(b, V, V1, m)$
$\texttt{IsOverridden}(a, n) \mapsto \texttt{IsOverridden}(a, m)$
$\texttt{IsOverridden}(b, n) \mapsto \texttt{IsOverridden}(b, m)$

Création de la délégation La création de la délégation est la finalité de cette opération et ceci est décrite par la règle suivante :
$\texttt{IsDelegator}(s, m, n) \mapsto \top$

Héritage des propriétés par le déléguée. La méthode `n` qui sera le déléguée de la méthode `m` héritera de cette méthode quelques propriétés sans que ces propriétés soient perdues par `m`. Ceci est décrit par exemple par la règle suivante :
$\texttt{IsDelegator}(a, n, V) \mapsto \texttt{IsDelegator}(a, m, V)$

$\texttt{IsDelegator}(b, n, V) \mapsto \texttt{IsDelegator}(b, m, V)$

Cette rétro-description veut dire par exemple que si la méthode m est un délégateur de n'importe quelle autre méthode (exprimé par la méta variable V), alors la méthode n le sera aussi après l'application de l'opération *CreateIndirectionInSuperclass*. Voici ci-dessous d'autres rétro-descriptions qui décrivent l'héritage des propriétés par n :

$\texttt{BoundVariableInMethodBody}(a, n, V) \mapsto \texttt{BoundVariableInMethodBody}(a, m, V)$

$\texttt{BoundVariableInMethodBody}(b, n, V) \mapsto \texttt{BoundVariableInMethodBody}(b, m, V)$

$\texttt{IsOverridden}(a, n) \mapsto \texttt{IsOverridden}(a, m)$

$\texttt{IsOverridden}(b, n) \mapsto \texttt{IsOverridden}(b, m)$

$\texttt{IsOverriding}(a, n) \mapsto \texttt{IsOverriding}(a, m)$

$\texttt{IsOverriding}(b, n) \mapsto \texttt{IsOverriding}(b, m)$

Passage des propriétés vers le déléguée Il s'agit de passage de quelques propriétés de la méthode délégateur m vers la méthode déléguée n après l'application de l'opération en question. Ceci est décrit par exemple par la rétro-descriptions suivante :

$\texttt{ExistsAbstractMethod}(s, n) \mapsto \texttt{ExistsAbstractMethod}(s, m)$

$\texttt{ExistsAbstractMethod}(s, m) \mapsto \bot$

Cette rétro-descriptions indique que la méthode m ne devient plus abstraite dans la classe s après l'application de l'opération vu qu'elle deviendra un délégateur, par contre la méthode n deviendra abstraite dans la classe s.

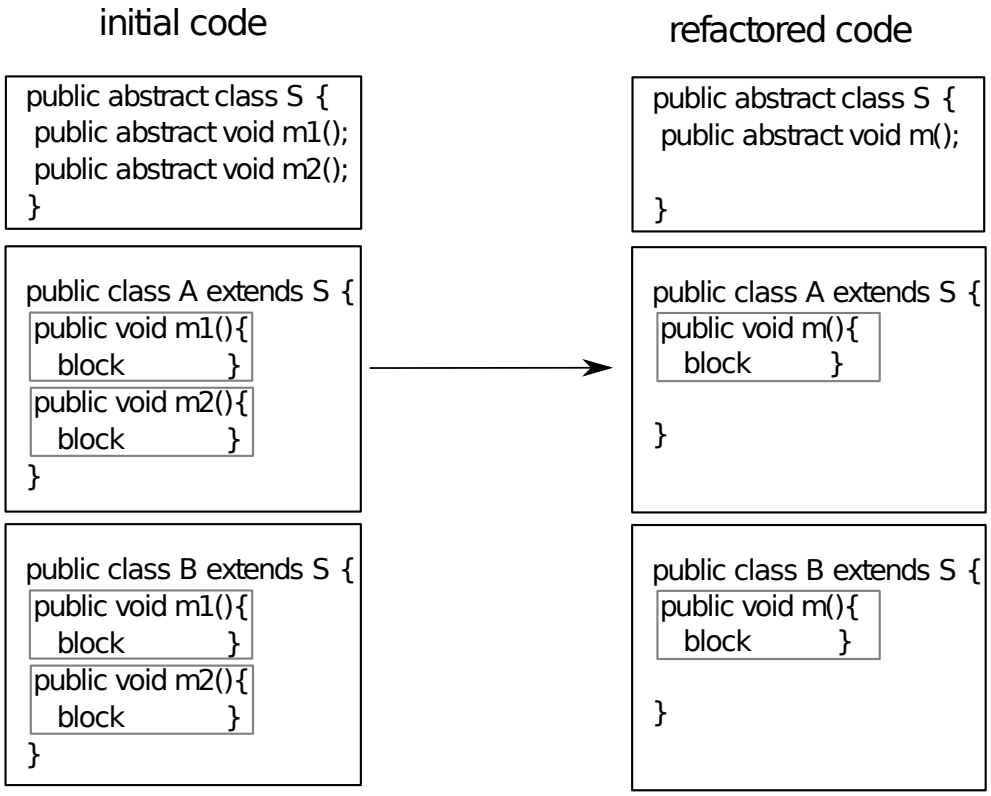
Remarque. La taille de la précondition et du rétro-description dépend de la taille des données qu'on fournit à l'opération de refactoring. Par exemple si on a dix sous classes de la classe S on aura 10 occurrences de ExistsClass (exemple : $\texttt{ExistsClass}(A) \wedge \texttt{ExistsClass}(B) \wedge ...$).

Opération composée : *MergeDuplicateMethods*

Cette opération est appliquée dans l'étape 8 de l'algorithme de transformation du Composite vers le Visiteur (Fig. 3, page 33).

Rôle

MergeDuplicateMethods (classname s, subclasses [a,b],mergedmethods [m,n],newmethod m2, invertedtype t, returntype q) : Cette opération est utilisée pour combiner deux méthodes *m* et *n* qui existent dans une hiérarchie de classes et qui sont égales de point de vue sémantique. La spécification formelle de cette opération se base sur la spécification des cinq opérations de refactoring qui la constituent.



Algorithme d'exécution

Cette opération s'exécute selon l'enchainement des opérations de refactoring élémentaires suivantes :

MergeDuplicateMethods (s,[a,b],[m,n],m2,t,q) =

1. ReplaceMethodcodeDuplicatesInverter (s, m, [n], t,q)
2. PullupConcreteDelegator(s, [a,b], n ,m)
3. InlineAndDelete(s,n)
4. RenameInHierarchyNoOverloading (s, [a,b], m,[t], m2)

les opérations qui constituent cette opération sont toutes offertes par l'outil de refactoring de IntelliJ IDEA que nous utilisons et sont décrites par leurs préconditions et rétro-descriptions dans l'annexe B.

Préconditions

Voici ci-dessous la précondition générée par notre calcul et qui vérifie statiquement la réussite de l'opération composée *MergeDuplicateMethods*. Nous ne nous intéressons pas à l'interprétation de ce résultat : nous interpréterons plus tard les préconditions de la totalité de la transformation (section 6). Nous présentons ce résultat à titre d'illustration.

$$
\begin{aligned}
&\texttt{ExistsMethodDefinition}(s, m)\\
&\wedge\, \texttt{ExistsMethodDefinitionWithParams}(s, m, [t])\\
&\wedge\, \texttt{AllSubclasses}(s, [a; b])\\
&\wedge\, \neg\texttt{ExistsMethodDefinition}(s, m2)\\
&\wedge\, \neg\texttt{ExistsMethodDefinition}(a, m2)\\
&\wedge\, \neg\texttt{ExistsMethodDefinition}(b, m2)\\
&\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(s, m2, [t])\\
&\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(a, m2, [t])\\
&\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(b, m2, [t])
\end{aligned}
$$

$\wedge \neg$IsOverloaded(s, m)
$\wedge \neg$IsOverloaded(a, m)
$\wedge \neg$IsOverloaded(b, m)
$\wedge \neg$IsInheritedMethod$(s, m2)$
$\wedge \neg$IsOverriding(s, n)
$\wedge \neg$IsOverridden(s, n)
$\wedge \neg$IsRecursiveMethod(s, n)
$\wedge \neg$IsUsedMethodIn(s, n, a)
$\wedge \neg$IsUsedMethodIn(s, n, b)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded$(b, n, this)$
$\wedge$ ExistsClass(s)
$\wedge$ IsAbstractClass(s)
$\wedge$ ExistsAbstractMethod(s, n)
$\wedge$ AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC$(a, n, this, s)$
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded$(a, n, this)$
$\wedge \neg$IsPrivate(a, n)
$\wedge$ ExistsClass(b)
$\wedge$ ExistsMethodDefinition(b, m)
$\wedge$ ExistsMethodDefinition(b, n)
$\wedge$ ExistsClass(a)
$\wedge$ ExistsMethodDefinition(a, m)
$\wedge$ ExistsMethodDefinition(a, n)
$\wedge$ IsInverter(b, m, t, q)
$\wedge$ IsInverter(b, n, t, q)
$\wedge$ IsInverter(a, m, t, q)
$\wedge$ IsInverter(a, n, t, q)

Le but de cette opération est de combiner deux méthodes qui sont égales de point de vue sémantique, ce qui est vérifié dans cette précondition par la proposition IsInverter qui est déjà décrit dans la section 2.

Remarque. L'utilisation des opérations composées a pour but de structurer les algorithmes de transformation. Ceci dit leur utilisation n'est pas obligatoire car nous pouvons directement appeler les opérations sur lesquelles elles se fondent. Nous verrons dans la section 6 que ces opérations ne figurent plus dans les chaines des opérations lorsque nous appliquons nos algorithmes de transformation. Ceci est due au fait que les opérations composées se déplient vers les opérations qui la constituent lorsque nous appliquons la transformation.

5 Validation de la description des opérations de refactoring

Nous fournissons une description pour chaque opération de refactoring (précondition plus rétro-description). Ces descriptions sont validées par l'application du calcul de la précondition à des exemples : transformation Composite-Visiteur de base aller-retour (chapitre 3), variations (chapitre 5) et une étude de cas (chapitre 6). Il s'agit donc d'une *validation par le test*. Cette validation reste donc limitée dans un contexte scientifique (d'autant plus qu'on utilise le calcul de préconditions pour valider ces transformations – la validation se mord donc la queue).

Pour pallier à ce problème, on peut s'appuyer sur des validations formelles (par preuve) d'opérations de refactoring. Des travaux existent dans ce domaine [LT05], mais sont actuellement peu nombreux (peu d'opérations sont couvertes). Ce travail dépasse le cadre de cette thèse.

6 Précondition pour la transformation entre un Composite et un Visiteur

Nous appliquons le système de calcul alimenté par nos préconditions et rétro-descriptions sur la transformation Composite ↔ Visiteur que nous avons proposée dans le chapitre 3. Ceci nous permet de générer les préconditions qui garantissent statiquement le non échec et la réversibilité de cette transformation.

Chaines d'opérations de refactoring

Nous fournissons les informations nécessaires sur le programme initial et sur le programme cible au système de calcul. Les algorithmes 5 et 6 sont les chaines des opérations de refactoring respectives pour l'instanciation des algorithmes de transformation Composite → Visiteur et Visiteur → Composite sur le programme 7 présenté dans le chapitre 3.

Nous remarquons l'absence des opérations de refactoring composées l'opération *MergeDuplicateMethods*. Ces opérations sont remplacées par leurs constituants.

```
CreateEmptyClass(PrintVisitor) ;
CreateEmptyClass(PrettyprintVisitor) ;
CreateIndirectionInSuperClass(Graphic,[Ellipse ;Composite], print, [], void, printAux) ;
CreateIndirectionInSuperClass(Graphic, [Ellipse ;Composite], show, [], void, showAux) ;
InlineMethodInvocations(Composite, printAux, [], Graphic, print, []) ;
InlineMethodInvocations(Composite, showAux, [], Graphic, show, []) ;
AddParameterWithReuse(Graphic, [Ellipse ;Composite], printAux, [], PrintVisitor, v) ;
AddParameterWithReuse(Graphic, [Ellipse ;Composite], showAux, [], PrettyprintVisitor, v) ;
MoveMethodWithDelegate (Ellipse, [graphics], PrintVisitor, printAux, [PrintVisitor], void, visit) ;
MoveMethodWithDelegate (Composite, [graphics], PrintVisitor, printAux, [PrintVisitor], void, visit) ;
MoveMethodWithDelegate (Ellipse, [graphics], PrettyprintVisitor, showAux, [PrettyprintVisitor], void, visit) ;
MoveMethodWithDelegate (Composite, [graphics], PrettyprintVisitor, showAux, [PrettyprintVisitor], void, visit) ;
ExtractSuperClass ([PrintVisitor ;PrettyprintVisitor], Visitor, [visit], void) ;
GeneraliseParameter (Graphic, [Ellipse ;Composite], printAux, v, PrintVisitor, Visitor) ;
GeneraliseParameter (Graphic, [Ellipse ;Composite], showAux, v, PrettyprintVisitor, Visitor) ;
ReplaceMethodcodeDuplicatesInverter (Ellipse, printAux, [showAux], Visitor, void) ;
ReplaceMethodcodeDuplicatesInverter (Composite, printAux, [showAux], Visitor, void) ;
PullupConcreteDelegator (Ellipse, [graphics], showAux, Graphic) ;
SafeDeleteWithOverridden (Composite, showAux, Graphic) ;
InlineAndDelete (Graphic, showAux) RenameInHierarchyNoOverloading (Graphic,[Ellipse ;Composite], printAux,[Visitor], accept)
```

Algorithme 5: Chaine d'opérations de refactoring de la transformation Composite→Visiteur instanciée de l'algorithme 3 (voir page 44) et appliquée sur le programme 7 (voir page 41)

Précondition générée

La formule 1 (page 77) montre la liste des préconditions générées par le système de calcul des préconditions pour le point fixe de la transformation (la précondition générée reste la même indépendamment du nombre d'aller/retour qu'on effectue et garantit que la transformation est valable pour la structure de départ et d'arrivée indifféremment de ces itérations d'aller/retour). Nous allons interpréter quelques préconditions pour montrer la consistance du résultat.

Interprétation de la précondition générée

Parmi les préconditions figurant dans la formule 1, notons :

La non-existence de la méthode *accept*. La transformation doit vérifier que la méthode *accept* n'existe pas dans le programme structuré selon le patron Composite :

DuplicateMethodInHierarchy (Graphic, [Ellipse ;Composite], accept,[visit], [print ;show], acceptPrintVisitoraddspecializedMethodtmp, [Visitor]) ;
SpecialiseParameter (Graphic, [Ellipse ;Composite], acceptPrintVisitoraddspecializedMethodtmp, Visitor, [PrintVisitor ;PrettyprintVisitor], PrintVisitor) ;
RenameDelegatorWithOverloading (Graphic, [Ellipse ;Composite], acceptPrintVisitoraddspecializedMethodtmp, PrintVisitor, v, Visitor, accept) ;
DuplicateMethodInHierarchy (Graphic, [Ellipse ;Composite], accept, [visit], [print ;show], acceptPrettyprintVisitoraddspecializedMethodtmp, [Visitor]) ;
SpecialiseParameter (Graphic, [Ellipse ;Composite], acceptPrettyprintVisitoraddspecializedMethodtmp, Visitor, [PrintVisitor ;PrettyprintVisitor], PrettyprintVisitor) ;
RenameDelegatorWithOverloading (Graphic, [Ellipse ;Composite], acceptPrettyprintVisitoraddspecializedMethodtmp, PrettyprintVisitor, v, Visitor, accept) ;
DeleteMethodInHierarchy (Graphic, [Ellipse ;Composite], accept, [visit], Visitor) PushDownAll (Visitor, [PrintVisitor ;PrettyprintVisitor], visit, [Ellipse]) ;
PushDownAll (Visitor, [PrintVisitor ;PrettyprintVisitor], visit, [Composite]) ;
InlineMethod (Ellipse, visit, PrintVisitor, accept) ;
InlineMethod (Composite, visit, PrintVisitor, accept) ;
InlineMethod (Ellipse, visit, PrettyprintVisitor, accept) ;
InlineMethod (Composite, visit, PrettyprintVisitor, accept) ;
RenameOverloadedMethodInHierarchy (Graphic, [Ellipse ;Composite], accept, printAux, [PrintVisitor]) ;
RenameOverloadedMethodInHierarchy (Graphic, [Ellipse ;Composite], accept, showAux, [PrettyprintVisitor]) ;
RemoveParameter (Graphic, [Ellipse ;Composite], printAux, [PrintVisitor], PrintVisitor, v) ;
RemoveParameter (Graphic, [Ellipse ;Composite], showAux, [PrettyprintVisitor], PrettyprintVisitor, v) ;
ReplaceMethodDuplication (Graphic, [Ellipse ;Composite], print, printAux, []) ;
ReplaceMethodDuplication (Graphic, [Ellipse ;Composite], show, showAux, []) ;
PushDownImplementation (Graphic, [], [Ellipse ;Composite], print, []) ;
PushDownImplementation (Graphic, [], [Ellipse ;Composite], show, []) ;
PushDownAll (Graphic, [Ellipse ;Composite], printAux, []) ;
PushDownAll (Graphic, [Ellipse ;Composite], showAux, []) ;
InlineAndDelete (Ellipse, printAux) ;
InlineAndDelete (Composite, printAux) ;
InlineAndDelete (Ellipse, showAux) ;
InlineAndDelete (Composite, showAux) ;
DeleteClass (PrintVisitor, Visitor, [Ellipse ;Composite ;PrintVisitor ;PrettyprintVisitor ;Visitor ;
Graphic], [visit], [accept ;eval ;show]) ;
DeleteClass (PrettyprintVisitor, Visitor, [Ellipse ;Composite ;PrintVisitor ;PrettyprintVisitor ;Visitor ;
Graphic], [visit], [accept ;eval ;show]) ;
DeleteClass (Visitor, java.lang.Object,[Ellipse ;Composite ;PrintVisitor ;PrettyprintVisitor ;Visitor ;
Graphic], [visit], [accept ;eval ;show])

Algorithme 6: Chaine d'opérations de refactoring de la transformation Visiteur→Composite instanciée de l'algorithme 4 (voir page 49) et appliquée sur le programme 8 (voir page 42)

$\neg\texttt{ExistsMethod}(Graphic, accept)$
$\wedge\ \neg\texttt{ExistsMethod}(Ellipse, accept)$
$\wedge\ \neg\texttt{ExistsMethod}(Composite, accept)$
$\wedge\ \neg\texttt{IsInheritedMethod}(Graphic, accept)$
$\wedge\ \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(Composite, show, this)$
$\wedge\ \texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(Ellipse, show, this, Graphic)$
$\wedge\ \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(Ellipse, show, this)$
$\wedge\ \neg\texttt{ExistsType}(Visitor)$
$\wedge\ \texttt{ExistsClass}(Ellipse)$
$\wedge\ \neg\texttt{BoundVariableInMethodBody}(Graphic, show, v)$
$\wedge\ \neg\texttt{BoundVariableInMethodBody}(Graphic, print, v)$
$\wedge\ \texttt{IsRecursiveMethod}(Composite, show)$
$\wedge\ \texttt{ExistsClass}(Composite)$
$\wedge\ \texttt{IsRecursiveMethod}(Composite, print)$
$\wedge\ \texttt{ExistsMethodWithParams}(Graphic, show, [\,])$
$\wedge\ \texttt{ExistsAbstractMethod}(Graphic, show)$
$\wedge\ \neg\texttt{IsInheritedMethod}(Graphic, showAux)$
$\wedge\ \neg\texttt{IsInheritedMethodWithParams}(Graphic, showAux, [\,])$
$\wedge\ \neg\texttt{ExistsMethodWithParams}(Graphic, showAux, [\,])$
$\wedge\ \texttt{HasReturnType}(Graphic, show, void)$
$\wedge\ \texttt{ExistsMethod}(Graphic, show)$
$\wedge\ \texttt{ExistsMethod}(Ellipse, show)$
$\wedge\ \texttt{ExistsMethod}(Composite, show)$
$\wedge\ \neg\texttt{ExistsMethod}(Graphic, showAux)$
$\wedge\ \neg\texttt{ExistsMethod}(Ellipse, showAux)$
$\wedge\ \neg\texttt{ExistsMethod}(Composite, showAux)$
$\wedge\ \texttt{ExistsClass}(Graphic)$
$\wedge\ \texttt{IsAbstractClass}(Graphic)$
$\wedge\ \texttt{ExistsMethodWithParams}(Graphic, print, [\,])$
$\wedge\ \texttt{ExistsAbstractMethod}(Graphic, print)$
$\wedge\ \neg\texttt{IsInheritedMethod}(Graphic, printAux)$
$\wedge\ \neg\texttt{IsInheritedMethodWithParams}(Graphic, printAux, [\,])$
$\wedge\ \neg\texttt{ExistsMethodWithParams}(Graphic, printAux, [\,])$
$\wedge\ \texttt{AllSubclasses}(Graphic, [Ellipse; Composite])$
$\wedge\ \texttt{HasReturnType}(Graphic, print, void)$
$\wedge\ \neg\texttt{IsPrivate}(Graphic, print)$
$\wedge\ \neg\texttt{IsPrivate}(Ellipse, print)$
$\wedge\ \neg\texttt{IsPrivate}(Composite, print)$
$\wedge\ \texttt{ExistsMethod}(Graphic, print)$
$\wedge\ \texttt{ExistsMethod}(Ellipse, print)$
$\wedge\ \texttt{ExistsMethod}(Composite, print)$
$\wedge\ \neg\texttt{ExistsMethod}(Graphic, printAux)$
$\wedge\ \neg\texttt{ExistsMethod}(Ellipse, printAux)$
$\wedge\ \neg\texttt{ExistsMethod}(Composite, printAux)$
$\wedge\ \neg\texttt{ExistsType}(PrettyprintVisitor)$
$\wedge\ \neg\texttt{ExistsType}(PrintVisitor)$

Formule 1: Précondition minimale générée garantissant le *point fixe* d'une transformation $[Composite \rightarrow Visiteur; Visiteur \rightarrow Composite]$ pour le programme 7

$\neg$ExistsMethod$(Graphic, accept)$
$\wedge \neg$ExistsMethod$(Ellipse, accept)$
$\wedge \neg$ExistsMethod$(Composite, accept)$
$\wedge \neg$IsInheritedMethod$(Graphic, accept)$

La non-existence des méthodes auxiliaires. Ceci permet d'éviter un conflit de noms :

$\neg$ExistsMethod$(Graphic, printAux)$
$\wedge \neg$ExistsMethod$(Ellipse, printAux)$
$\wedge \neg$ExistsMethod$(Composite, printAux)$

Ces préconditions vérifient que les méthodes auxiliaires qui vont être crées lors de l'application de l'opération *CreateIndirectionInSuperClass* de l'étape 2 de l'algorithme de base n'existent pas dans le programme initial.

La non-existence des classes Visiteurs Cette propriété est vérifiée par les préconditions suivantes :
$\neg$ExistsType$(Visitor)$
$\wedge \neg$ExistsType$(PrettyprintVisitor)$
$\wedge \neg$ExistsType$(PrintVisitor)$

Ces préconditions sont nécessaires pour le bon déroulement des deux étapes 1 et 6 de l'algorithme 3 qui doivent vérifier respectivement la non-existence des classes PrintVisitor et ShowVisitor et la non existence de la classe Visitor.

Propriétés des méthodes métier. Certaines propriétés doivent être vérifiées par les méthodes métier du programme (dans l'exemple les méthodes print et show). Le premier paquet de ces propriétés est exprimé par :

AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded$(Composite, show, this)$
$\wedge$ AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC$(Ellipse, show, this, Graphic)$
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded$(Ellipse, show, this)$

Il s'agit de vérifier que dans le corps des deux méthodes show il n' y a aucune méthode qui est invoquée sur this. Ceci est demandé par l'opération composée *MergeDuplicateMethods* lorsque la méthode show doit être montée vers la classe Graphic.

Une autre vérifie l'existence de ces méthodes :

ExistsMethod$(Graphic, print)$
$\wedge$ ExistsMethod$(Ellipse, print)$
$\wedge$ ExistsMethod$(Composite, print)$
$\wedge$ ExistsMethod$(Graphic, show)$
$\wedge$ ExistsMethod$(Ellipse, show)$
$\wedge$ ExistsMethod$(Composite, show)$

Existence d'une hiérarchie de classes dont sa racine est une classe abstraite. Cette propriété est vérifiée par la précondition suivante :

ExistsClass$(Graphic)$
$\wedge$ IsAbstractClass$(Graphic)$
$\wedge$ AllSubclasses$(Graphic, [Ellipse; Composite])$

La formule 1 est vérifiée valable sur le programme 7, ce qui garantit *statiquement* que notre transformation assure la propriété *non-échec* et la réversibilité.

7 Bilan

Nous avons présenté dans ce chapitre comment nous construisons toutes les données nécessaires pour alimenter un système qui calcule les préconditions qui garantissent statiquement le non échec, la réversibilité et la préservation de la sémantique d'une composition d'opérations de refactoring et appliqué dans une transformation Composite $\leftrightarrow$ Visiteur. Nous avons décrit formellement chaque opération inclue dans nos transformations par une précondition et une rétro-description. Le total des règles de calcul basées sur ces préconditions et rétro-descriptions de 28 opérations est de l'ordre de 480 règles. Des vérifications restent à faire pour prouver que la spécification que nous définissons est consistante et complète.

5

Influence des variations des patrons Composite et Visiteur sur la transformation

Sommaire

Dans ce chapitre, nous traitons la transformation décrite dans le chapitre 3 pour quelques variations dans l'implémentation des patrons Composite et Visiteur. Nous avons pu identifier jusqu'à maintenant quatre variations de ces deux patrons. L'identification de ces quatre variations est apparue lors de la validation de la transformation de base sur JhotDraw. Nous avons découvert que la transformation de base doit être adaptée pour pouvoir être appliquée sur JhotDraw. Nous montrons dans ce chapitre comment adapter la transformation de base pour prendre en compte de chaque variation et nous verrons dans le chapitre 6 comment appliquer une transformation Composite $\leftrightarrow$ Visiteur sur les quatre variations à la fois. Les quatre variations identifiées concerne le patron Composite et ont les caractéristiques suivantes :

- Les méthodes métier portent des paramètres,
- Les méthodes métier ont des types de retour différents,
- La hiérarchie du Composite a plusieurs niveaux avec une répartition aléatoire des définitions des méthodes métier dans les sous classes (on peut trouver des méthodes redéfinies et des méthodes héritées mais non redéfinies),
- La racine de la hiérarchie du Composite est une interface et non pas une classe abstraite.

1 Méthodes avec paramètres

Nous considérons que certaines méthodes métiers ont des paramètres et d'autres non. Cette petite variation exige l'adaptation des algorithmes de base (algorithmes 3 et 4) . Nous verrons dans cette section les algorithmes dédiés pour la transformation réversible entre le Composite et le Visiteur de cette variation. Pour distinguer les méthodes avec paramètres et celles sans paramètres nous utilisons les notations suivantes :

Notation	Définition
$\mathbb{M}_{\mathbb{P}}$	Ensemble des méthodes avec paramètres
$\mathbb{M}_{\mathbb{W}}$	Ensemble des méthodes sans paramètres, avec $\mathbb{M}_{\mathbb{P}} \cup \mathbb{M}_{\mathbb{W}} = \mathbb{M}$ et $\mathbb{M}_{\mathbb{P}} \cap \mathbb{M}_{\mathbb{W}} = \emptyset$

Structure Composite

Nous considérons une méthode print sans paramètres et une méthode setcolor avec paramètres dans le programme. 9 qui est structuré selon le patron Composite. Dans ce programme nous répartissons les méthodes comme suit :

- $\mathbb{M}_{\mathbb{W}}$ = {print }
- $\mathbb{M}_{\mathbb{P}}$ = {setColor }

La méthode setColor a un paramètre c de type int. Ce paramètre est passée pour chaque appel récursif dans la classe Composite de la façon suivante :

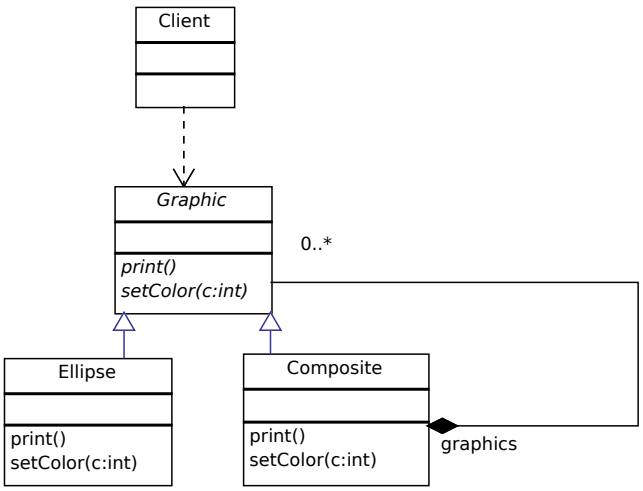
```
abstract class Graphic{
 abstract void setColor(int c);
 ...
}
class Composite extends Graphics{

 void setColor(int c) {
   for (Graphic g :graphics){
      g.setColor(c);}
 }
 ...
}
```

Structure Visiteur

Nous ciblons par notre transformation de passer du programme 9 qui est structuré selon le patron Composite selon cette variation vers le programme 10 qui est structuré selon le patron Visiteur.

Dans la structure Visiteur du programme 9, l'objet de type Visitor, qui est créé par les méthodes métiers de la racine Graphic de la partie Composite, est passée d'une manière récursive comme paramètre pour la méthode *accept* et comme un objet receveur des appels de la méthode *visit*. C'est grâce à ceci, que le paramètre c peut être mis dans l'état de cet objet Visiteur pour qu'il soit disponible lors du double *aiguillage* entre la partie Composite et la partie Visiteur comme indiqué dans le code source suivant :

```
class SetColorVisitor extends Visitor{

 private final int c;

 SetColorVisitor (int c){ this.c = c ; }

 void visit(Ellipse e){ e.color = c; }

 void visit(Composite c){
  for(Graphic g : c.graphics) {g.accept(this);} }
}
```




```
abstract class Graphic {

  public abstract void print();

  public abstract void setColor(int c);

}
```

```
class Ellipse extends Graphic{

 protected int color ;

 public void print() {
  System.out.println("Ellipse with color:"+ color);
 }

 public void setColor(int c){
     this.color = c;
 }
}
```

```
class Composite extends Graphic{

 List<Graphic> graphics = new ArrayList<Graphic>();

 public void print() {
   System.out.println("Composite:");
   for (Graphic g : graphics) {
            g.print();
   }
 }

 public void setColor(int c){
   for (Graphic g : graphics) {
            g.setColor(c);
   }
 }


}
```

Programme 9: Composite avec des méthodes avec paramètres.

En fait, la méthode setColor de la classe abstraite Graphic passe le paramètre c au constructeur de la classe SetColorVisitor, puis il passe l'instance de cette classe (avec la paramètre c dans son état) à la méthode *accept* (voir le code source ci-dessous). Dans la classe Graphic :

```
void setColor(int c) { accept(new SetColorVisitor(c)); }
```

Dans les classes Ellipse et Composite, la méthode *accept* garde la même définition que celle de la structure Visiteur de base.

Transformation du Composite vers le Visiteur

L'algorithme 7 montre les adaptations que nous devons faire pour réaliser une transformation d'un Composite comportant des méthodes avec paramètres vers le Visiteur correspondant. Nous distinguons les nouvelles opérations de refactoring par la police texte gras :

L'étape 4 de l'algorithme 3 de base, est appliquée pour ajouter des paramètres de type Visiteur aux méthodes temporaires qui vont devenir plus tard des méthodes *accept*. Cette étape est valable uniquement pour l'ensemble $\mathbb{M}_{\mathbb{W}}$ des méthodes sans paramètres. Pour les méthodes avec paramètres (l'ensemble $\mathbb{M}_{\mathbb{P}}$), nous voudrons remplacer les paramètres de ces méthodes par un seul paramètre de type Visiteur afin d'avoir plus tard des méthodes *accept* avec un seul paramètre. Pour réaliser ceci, nous appliquons l'opération *Introduce Parameter Object* (étape 4.A de l'algorithme 7). Cette opération remplace aussi l'étape 1 de l'algorithme de base car elle sert aussi à créer une nouvelle classe qui correspond au type de paramètre ajouté.

Introduire un *parameter object* de type A à la méthode m(B b) par exemple crée une classe A, déplace le paramètre b vers A comme une variable d'instance et finalement, change la méthode m(B b) en m(A a) (tout ancien accès à b dans le corps de la méthode m sera remplacé par a.b.

Pour les méthodes sans paramètres, nous appliquons l'étape 1.A et la deuxième sous étape de l'étape 4.A. Ces deux étapes jouent respectivement le rôle des étapes 1 et 4 de l'algorithme de base et elles sont déjà détaillées dans le chapitre 3 (page 44).

Le Programme 11 montre l'état du programme initial après l'application de l'étape 4. A. Á ce stade, le programme est prêt pour déplacer le code métier de la partie Composite vers la partie Visiteur.

Après le déplacement du code métier de la partie Composite vers la partie Visiteur (étape 5 de l'algorithme de base) le paramètre c est mis dans l'état de l'objet Visiteur pour qu'il soit disponible lors du double *aiguillage* entre les deux parties du Visiteur comme suit :

```
abstract class Graphic{
 ...
 public void setColor(int c){
     setColorAux(new SetColorVisitor(c));
 }
 abstract void setColorAux(SetColorVisitor v);

}



class Composite extends Graphic{
 ...
 public void setColorAux(SetColorVisitor v){
   v.visit(this);
 }
 ...
}
```

```
abstract class Graphic {

  public void print(){
    accept(new PrintVisitor());
  }
  public void setColor(int c) {
    accept(new SetColorVisitor(c));
  }
  public abstract void accept(Visitor v);
}
```

```
class Ellipse extends Graphic {
 protected int color ;

 void accept(Visitor v) {
        v.visit(this);  }
}
```

```
class Composite extends Graphic{

 List<Graphic> graphics = new ArrayList<Graphic>();

 public void accept(Visitor v){
        v.visit(this);
 }

}
```

```
public class PrintVisitor extends Visitor {

    public void visit(Composite c) {
        System.out.println("Composite:");
        for (Graphic g : c.graphics) {
            g.accept(this);
        }
    }
    public void visit(Ellipse e) {
        System.out.println("Ellipse with color:"+ e.color); }
}
```

```
public class SetColorVisitor extends Visitor{
    private final int c;
    public SetColorVisitor(int c) {
        this.c = c;              }
    public int getC() {
        return c;     }

    public void visit(Composite c) {
        for (Graphic g : c.graphics) {
            g.accept(this);
        }
    }
    public void visit(Ellipse e) {
        e.color = getC();
    }
}
```

```
public abstract class Visitor {

 public abstract void visit(Composite c);
 public abstract void visit(Ellipse e);
}
```

Programme 10: Visiteur correspondant à un Composite comportant des méthodes avec paramètres (partie visiteur)

1.A) ForAll m in $\mathbb{M}_{\mathbb{W}}$ do *CreateEmptyClass*(vis(m))	(remplace l'étape 1)
4.A) ForAll m in $\mathbb{M}_{\mathbb{P}}$ do ***IntroduceParameterObject***(S, aux(m), vis(m))	
ForAll m in $\mathbb{M}_{\mathbb{W}}$ do *AddParameterWithReuse*(S, aux(m), vis(m))	(remplace l'étape 4)

Algorithme 7: Algorithme de la transformation du Composite vers le Visiteur de la variation méthodes avec paramètres

```
class SetColorVisitor{

    public void visit(Composite c) {
        for (Graphic g : c.graphics) {
            g.setColorAux(new SetColorVisitor(getC()));
        }
    }
}
```

Pour le reste des étapes nous gardons les mêmes étapes de l'algorithme 3 de base.

Transformation du Visiteur vers le Composite

L'algorithme 8 montre les adaptations que nous devons faire pour assurer la transformation d'un Visiteur vers un Composite comportant des méthodes avec paramètres :

La tâche qui fait la différence entre l'algorithme de base et cette transformation est la préparation du programme pour supprimer les classes Visiteurs . En fait, pour supprimer une classe, nous devons vérifier qu'il n'y a aucune référence vers cette classe. Cette tâche est effectuée en deux endroits différents selon que les méthodes métier ont des paramètres ou non comme suit :

- Pour les méthodes sans paramètres, nous pouvons directement supprimer les paramètres de type Visiteurs à l'étape VI.A pour se débarrasser de toute référence vers les classes Visiteurs afin de pouvoir les supprimer plus tard.

  En fait, après l'application de l'étape IV, les classes Visiteurs n'ont aucune référence au niveau des méthodes sans paramètres sauf leur existence comme paramètres pour ces méthodes et par la suite, nous pouvons appliquer directement l'étape VI.A. Le code source suivant montre que de la méthode printAux (la méthode déléguée de la méthode print()) utilise la classe PrintVisitor uniquement comme paramètre et elle ne l'utilise pas dans son corps après l'application de l'étape IV :

  ```
  class Ellipse extends Graphic
   void printAux(PrintVisitor v){
         System.out.println("Ellipse with color"+ color);
   }
   ...
  }
  ```

- Pour les méthodes avec paramètres, nous ne pouvons pas supprimer le paramètre de type Visiteur à ce stade car leurs déléguées utilisent encore l'objet Visiteur pour accéder au champs c de cette classe. C'est le cas de la méthode setColorAux qui a l'état suivant à ce stade :

  ```
  class Ellipse extends Graphic{
    void setColorAux(SetColorVisitor v){
  ```

```
abstract class Graphic {

    public void print() {
        printAux(new PrintVisitor());
    }

    public abstract void printAux(PrintVisitor v);

    public void setColor(int c) {
        setColorAux(new SetColorVisitor(c));
    }

    public abstract void setColorAux(SetColorVisitor v);

}
```

```
class Ellipse extends Graphic{

 protected int color ;

 public void printAux(PrintVisitor v) {
  System.out.println("Ellipse with color:"+ color);
 }

 public void setColorAux(SetColorVisitor v){
     this.color = v.getC();
 }
}
```

```
class Composite extends Graphic{

 List<Graphic> graphics = new ArrayList<Graphic>();

 public void printAux(PrintVisitor v) {
   System.out.println("Composite:");
   for (Graphic g :graphics) {
       g.printAux(v);
   }
 }

 public void setColorAux(SetColorVisitor v){
   for (Graphic g :graphics) {
       g.setColorAux(new SetColorVisitor(v.getC()));
   }
 }


}
```

```
public class SetColorVisitor {
    private final int c;

    public SetColorVisitor(int c) {
        this.c = c;
    }

    public int getC() {
        return c;
    }
}
```

```
public class PrintVisitor {
}
```

Programme 11: État du Composite avec des méthodes avec paramètres avant le déplacement du code métier (partie Visiteur).

VI.A ForAll m in $\mathbb{M}_{\mathbb{W}}$ do *RemoveParameter*(S,aux(m),vis(m))	(remplace l'étape VI)
XI.A ForAll m in $\mathbb{M}_{\mathbb{P}}$ do **InlineParameterObject**(S, aux(m), vis(m))	(avant l'étape XI)

Algorithme 8: Algorithme de la transformation du Visiteur vers le Composite de la variation méthodes avec paramètres

```
        color = v.getC();
  }
  ...
}
```

Après l'application de l'étape X, la méthode setColor a l'état suivant :

```
class Ellipse extends Graphic{
  void setColor(int c){
        color = new SetColorVisitor(c).getC();
  }
  ...
}
```

Pour supprimer toute référence des classes Visiteurs au niveau des méthodes avec paramètres, nous appliquons l'opération *InlineParameterObject* (étape XI.A) qui va remplacer new SetColorVisitor(c).getC() par c comme suit :

```
class Ellipse exntends Graphic{
 void setColor(int c){
        color = c;
 }
 ...
}
```

Après l'application de l'étape XI.A, nous pouvons appliquer les étapes XI et XII pour supprimer la hiérarchie des classes Visiteurs.

Précondition générée et interprétations

Nous avons intégré les nouvelles opérations dans notre calcul des préconditions (voir annexe C). La précondition 1 vérifie le non-échec de l'algorithme 7. Voici ci-dessous les interprétations des différences qui caractérisent cette précondition par rapport à celle qui vérifie la transformation de base :

Vérification de la signature des méthodes avec paramètres Nous remarquons que certains prédicats affichent les paramètres des méthodes avec paramètres comme :
`ExistsMethodDefinitionWithParams`$(Graphic, setColor, [intc]$
$\wedge$ `BoundVariableInMethodBody`$(Ellipse, setColor, intc)$

Ces prédicats sont utilisés dans ce contexte pour vérifier que la méthode setColor a des paramètres. Ceci est demandée par l'opération *IntroduceParameterObject* (step 4.A) qui ne s'exécute qu'avec les méthodes avec paramètres. Le programme 9 qui est utilisé comme une entrée pour la transformation vérifie cette propriété.

Vérification de non existence des classes Visiteurs. Dans la précondition de la transformation de base, la vérification de non-existence des classes Visiteurs se fait dans un même endroit. C'est dans l'étape 1 que cette vérification se fait pour toutes les classes Visiteurs. Mais, dans cette transformation nous remarquons que les prédicats $\neg$`ExistsType`$(PrintVisitor)$ et $\neg$`ExistsType`$(SetColorVisitor)$ apparaissent dans deux endroits différents. Ceci est due au fait que la première précondition est vérifiée dans l'étape 1.A et concerne les méthodes sans paramètres et la deuxième est vérifiée dans l'étape 4.A lors de l'exécution de l'opération *IntroduceParameterObject*. Cette opération va ajouter un paramètre à la méthode en question et crée une classe qui représente le type de ce paramètre, ce qui explique que cette opération vérifie l'existence de cette classe (la classe SetColorVisitor dans l'exemple).

$(\neg\texttt{IsUsedConstructorAsMethodParameter}(SetColorVisitor, Ellipse, print)$
$\wedge \neg\texttt{IsUsedConstructorAsMethodParameter}(SetColorVisitor, Composite, print)$
$\wedge \neg\texttt{IsUsedConstructorAsMethodParameter}(SetColorVisitor, Graphic, print)$
$\wedge \neg\texttt{IsUsedConstructorAsMethodParameter}(SetColorVisitor, Graphic, setColor)$
$\wedge \neg\texttt{IsUsedConstructorAsObjectReceiver}(SetColorVisitor, Ellipse, printAux)$
$\wedge \neg\texttt{IsUsedConstructorAsObjectReceiver}(SetColorVisitor, Ellipse, print)$
$\wedge \neg\texttt{IsUsedConstructorAsObjectReceiver}(SetColorVisitor, Composite, printAux)$
$\wedge \neg\texttt{IsUsedConstructorAsObjectReceiver}(SetColorVisitor, Composite, print)$
$\wedge \texttt{ExistsType}(Composite)$
$\wedge \texttt{ExistsType}(Ellipse)$
$\wedge \neg\texttt{ExistsMethodInvocation}(Ellipse, setColorAux, Ellipse, print)$
$\wedge \neg\texttt{IsUsedMethod}(Graphic, accept_SetColorVisitor_addspecializedMethod_tmp, [SetColorVisitor])$
$\wedge \neg\texttt{ExistsMethodDefinition}(Graphic, accept)$
$\wedge \neg\texttt{ExistsMethodDefinition}(Ellipse, accept)$
$\wedge \neg\texttt{ExistsMethodDefinition}(Composite, accept)$
$\wedge \neg\texttt{IsInheritedMethod}(Graphic, accept)$
$\wedge \neg\texttt{IsUsedMethodIn}(Graphic, setColorAux, Ellipse)$
$\wedge \neg\texttt{IsUsedMethodIn}(Graphic, setColorAux, Composite)$
$\wedge \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(Composite, setColor, this)$
$\wedge \texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(Ellipse, setColor, this, Graphic)$
$\wedge \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(Ellipse, setColor, this)$
$\wedge \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(Graphic, setColor, v)$
$\wedge \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(Ellipse, setColor, v)$
$\wedge \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(Composite, setColor, v)$
$\wedge \neg\texttt{ExistsType}(Visitor)$
$\wedge \texttt{ExistsClass}(Ellipse)$
$\wedge \texttt{ExistsType}(Graphic)$
$\wedge \neg\texttt{ExistsType}(SetColorVisitor)$
$\wedge \texttt{BoundVariableInMethodBody}(Ellipse, setColor, intc)$
$\wedge \texttt{BoundVariableInMethodBody}(Composite, setColor, intc)$
$\wedge \neg\texttt{BoundVariableInMethodBody}(Graphic, print, v)$
$\wedge \texttt{IsRecursiveMethod}(Composite, setColor)$
$\wedge \texttt{ExistsClass}(Composite)$
$\wedge \texttt{IsRecursiveMethod}(Composite, print)$
$\wedge \texttt{ExistsMethodDefinitionWithParams}(Graphic, setColor, [intc])$
$\wedge \texttt{ExistsAbstractMethod}(Graphic, setColor)$
$\wedge \neg\texttt{IsInheritedMethod}(Graphic, setColorAux)$
$\wedge \neg\texttt{IsInheritedMethodWithParams}(Graphic, setColorAux, [intc])$
$\wedge \neg\texttt{ExistsMethodDefinitionWithParams}(Graphic, setColorAux, [intc])$
$\wedge \texttt{HasReturnType}(Graphic, setColor, void)$
$\wedge \texttt{ExistsMethodDefinition}(Graphic, setColor)$
$\wedge \texttt{ExistsMethodDefinition}(Ellipse, setColor)$
$\wedge \texttt{ExistsMethodDefinition}(Composite, setColor)$
$\wedge \neg\texttt{ExistsMethodDefinition}(Graphic, setColorAux)$
$\wedge \neg\texttt{ExistsMethodDefinition}(Ellipse, setColorAux)$
$\wedge \neg\texttt{ExistsMethodDefinition}(Composite, setColorAux)$
$\wedge \texttt{ExistsClass}(Graphic)$
$\wedge \texttt{IsAbstractClass}(Graphic)$
$\wedge \texttt{ExistsMethodDefinitionWithParams}(Graphic, print, [])$
$\wedge \texttt{ExistsAbstractMethod}(Graphic, print)$
$\wedge \neg\texttt{IsInheritedMethod}(Graphic, printAux)$
$\wedge \neg\texttt{IsInheritedMethodWithParams}(Graphic, printAux, [])$
$\wedge \neg\texttt{ExistsMethodDefinitionWithParams}(Graphic, printAux, [])$
$\wedge \texttt{AllSubclasses}(Graphic, [Ellipse; Composite])$
$\wedge \texttt{HasReturnType}(Graphic, print, void)$
$\wedge \neg\texttt{IsPrivate}(Graphic, print)$
$\wedge \neg\texttt{IsPrivate}(Ellipse, print)$
$\wedge \neg\texttt{IsPrivate}(Composite, print)$
$\wedge \texttt{ExistsMethodDefinition}(Graphic, print)$
$\wedge \texttt{ExistsMethodDefinition}(Ellipse, print)$
$\wedge \texttt{ExistsMethodDefinition}(Composite, print)$
$\wedge \neg\texttt{ExistsMethodDefinition}(Graphic, printAux)$
$\wedge \neg\texttt{ExistsMethodDefinition}(Ellipse, printAux)$
$\wedge \neg\texttt{ExistsMethodDefinition}(Composite, printAux)$
$\wedge \neg\texttt{ExistsType}(PrintVisitor))$

Precondition 1: Préconditions garantissant le non-échec de la transformation réversible de la variation mé-

2 Méthodes avec des types de retour différents

Dans cette section nous traitons un Composite dont les méthodes métiers retournent des valeurs de types différents.

Structure Composite

Nous considérons les deux méthodes *perimeter* et *toString* qui retournent respectivement *Integer* et *String*. Le programme 5.1 est un programme structuré selon le patron Composite avec des méthodes qui retournent des types différents.

Structure Visiteur

Le programme 12 est le programme qui correspond à la structure Visiteur du programme 5.1.

Nous remarquons que cette structure montre un patron Visiteur avec des types génériques. Ceci est due au fait que les méthodes de la structure initiale retournent des types différents et par la suite on ne peut pas utiliser void pour la méthode *accept* dans la structure finale comme nous l'avons déjà fait avec la transformation de base.

Parmi les solutions que nous pouvons envisager est de surcharger la méthode *accept* et créer une méthode *accept* pour chaque type de retour. Le programme 13 est structuré selon un patron Visiteur qui correspond au programme 5.1 mais sans types génériques.

De point de vue structurel, ceci dégrade la beauté du Visiteur : une méthode *accept* pour chaque méthode métier au lieu d'une seule méthode *accept* qui implémente la traversée entre le Composite et le Visiteur.

Pour obtenir un bon Visiteur, nous utilisons les Visiteurs génériques [OWG08].

Nous remarquons que chaque type de retour apparait comme un type paramétrique dans le Visiteur qui lui correspond. Chaque Visiteur représente une méthode métier ainsi que son type de retour comme indiqué ci-dessous.

```
class PerimeterVisitor extends Visitor <Integer> {...}

class ToStringVisitor extends Visitor <String> {...}
```

Remarque L'utilisation des types génériques est restrictive et ne supporte pas les types primitifs comme int ou bool. Dans ce cas, nous convertissons ces types primitifs vers des types classes (int et bool correspondent respectivement à Integer et Boolean). Nous traitons aussi les Composites dans lesquels on trouve des méthodes avec des types de retour et d'autres avec void qui sera convertit en : Object et en ajoutant une l'instruction return null ;

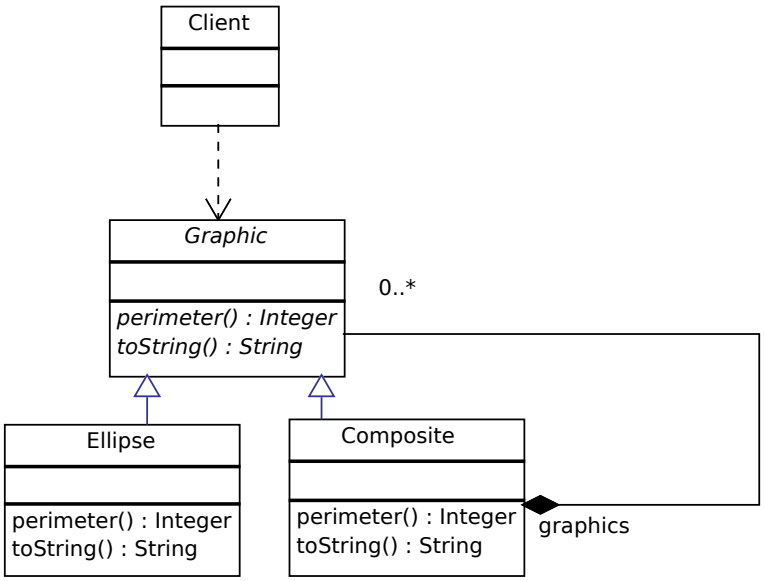
Transformation du Composite vers le Visiteur

Pour décrire l'algorithme de cette transformation nous ajoutons la notation suivante :

- $\mathbb{R}$: ensemble des méthodes et leurs types de retour correspondants, ici $\mathbb{R}$ ={(perimeter, Integer), (toString, String)}.

L'algorithme 9 montre les adaptations que nous devons faire pour réaliser une transformation d'un Composite comportant des méthodes retournant des valeurs différentes vers un Visiteur générique.

Dans l'étape 6 de l'algorithme de base, l'opération *ExtractSuperClass* est utilisé pour créer une nouvelle classe abstraite qui sera la classe mère des Visiteurs et créer des déclarations abstraites pour les méthodes *visit* au niveau de cette classe (ceci se fait par ce qu'on appelle un *pull-up* dans le refactoring). Dans cette variation, nous devons utiliser une extension de l'opération *pull up* qui permet d'introduire les types génériques dans la classe mère afin qu'elle puisse traiter les déclarations abstraites des méthodes *visit* qui ont des types de retour différent comme c'est montré par le code source suivant :



(a) Diagramme de classe.

```
abstract class Graphic{
    public abstract Integer perimeter ();
    public abstract String toString ();

}
```

```
class Ellipse extends Graphic{
  int perimeter;

  public Ellipse (int perimeter){
    this.perimeter=perimeter ;};

    public Integer perimeter () {
        return (perimeter);
    }

    public String  toString () {
        return ("Ellipse : " + Integer.toString(perimeter));
    }
}
```

```
class Composite extends Graphic {

  List<Graphic> graphics = new ArrayList<Graphic>();

  public Integer perimeter() {
   int acc = 0 ;
   for (Graphic g : graphics) {
      acc += g.perimeter();
   }
        return acc;
  }
  public String toString(){
   String s ;
   s = new String ("Composite with: ");
   for (Graphic g : graphics) {
      s = s.concat(g.toString() + ", ");
   }
   System.out.println("(end)");
   return s;
 }

}
```

(b) Code source Java.

FIGURE 5.1: Composite avec des méthodes retournant des types différents.

```
abstract class Graphic{
    public Integer perimeter() {
        return accept(new PerimeterVisitor());
    }
    public String toString() {
        return accept(new ToStringVisitor());
    }
    public abstract <T> T accept(Visitor<T> v);
}
```

```
class Ellipse extends Graphic{
  int perimeter;
  public Ellipse (int perimeter){
    this.perimeter=perimeter ;};

    public <T> T accept(Visitor<T> v) {
        return  v.visit(this);
    }
}
```

```
class Composite extends Graphic {
  List<Graphic> graphics = new ArrayList<Graphic>();

  public <T> T accept(Visitor<T> v) {
        return v.visit(this);
  }
}
```

```
public class PerimeterVisitor extends Visitor <Integer>{
  public Integer visit(Composite c) {
    int acc = 0 ;
    for (Graphic g : c.mChildGraphics) {
        acc += g.accept(this);
    }
    return acc;
  }
  public Integer visit(Ellipse e) {
        return (e.perimeter);
  }
}
```

```
public class ToStringVisitor extends Visitor <String> {
  public String visit(Composite c) {
   String s ;
   s = new String ("Composite with: ");
   for (Graphic g : c.graphics) {
       s = s.concat(g.accept(this) + ", ");
   }
   System.out.println("(end)");
      return s;
   }
    public String visit(Ellipse e) {
     return ("Ellipse : " + Integer.toString(e.perimeter));
    }
}
```

```
public abstract class Visitor <T>{
    public abstract T visit(Composite c);
    public abstract T visit(Ellipse e);
}
```

Programme 12: Visiteur avec des types génériques

```
abstract class Graphic{
    public Integer perimeter() {
        return accept(new PerimeterVisitor());
    }
    public abstract Integer accept(PerimeterVisitor v);

    public String toString() {
        return accept(new ToStringVisitor());
    }
    public abstract String accept(ToStringVisitor v);

}
```

```
class Ellipse extends Graphic {
  int perimeter;

  public Ellipse (int perimeter){
    this.perimeter=perimeter ;}

    public Integer accept(PerimeterVisitor v) {
        return v.visit(this);
    }
    public String accept(ToStringVisitor v) {
        return v.visit(this);
    }

}
```

```
class Composite extends Graphic {

    List<Graphic> graphics = new ArrayList<Graphic>();

    public Integer accept(PerimeterVisitor v) {
        return v.visit(this);
    }
    public String accept(ToStringVisitor v){
        return v.visit(this);
    }
}
```

```
public class Visitor {
}
```

```
public class PerimeterVisitor extends Visitor {
    public Integer visit(Composite c) {
        int acc = 0 ;
        for (Graphic g : c.graphics) {
            acc += g.accept(this);
        }
        return acc;
    }
    public Integer visit(Ellipse e) {
        return (e.perimeter);
    }
}
```

```
public class ToStringVisitor extends Visitor{
    public String visit(Composite c) {
        String s ;
        s = new String ("Composite with: ");
        for (Graphic g : c.graphics) {
        s = s.concat(g.accept(this) + ", ");
    }
        System.out.println("(end)");
        return s;
    }
    public String visit(Ellipse e) {
        return ("Ellipse : " + Integer.toString(e.perimeter));
    }
}
```

Programme 13: Visiteur sans types génériques

6.B **ExtractSuperClassWithoutPullUp**($\mathbb{V}$, "Visitor") ;
ForAll m in $\mathbb{M}$, c in $\mathbb{C}$ do **PullUpWithGenerics**(vis(m), "visit","Visitor") (remplace 6)

Algorithme 9: Algorithme de la transformation du Composite vers le Visiteur de la variation types génériques

```
class ToStringVisitor extends Visitor{

    public String visit(Composite c) {
        String s ;
        s = new String ("Composite with: ");
        for (Graphic g : c.graphics) {
        s = s.concat(g.toStringAux(this) + ", ");
    }
        System.out.println("(end)");
        return s;
    }

    public String visit(Ellipse e) {
        return ("Ellipse : " + Integer.toString(e.perimeter));
    }
}


class PerimeterVisitor extends Visitor {

   public Integer visit(Composite c) {
        int acc = 0 ;
        for (Graphic g : c.graphics) {
            acc += g.perimeterAux(this);
        }
        return acc;
    }

    public Integer visit(Ellipse e) {
        return (e.perimeter);
    }
}
```

Pour traiter cette variation, nous utilisons l'opération *ExtractSuperClassWithoutPullUp*, puis l'opération *PullUpWithGenerics*[1] au lieu de l'opération *ExtractSuperClass* appliquée dans l'étape 6 de l'algorithme de base. Ces deux opérations sont appliquées dans l'étape 6.B.

Transformation du Visiteur vers le Composite

L'algorithme 10 montre les adaptations que nous devons faire pour réaliser une transformation d'un Composite comportant des méthodes retournant des valeurs différentes vers un Visiteur générique.

Dans l'étape I de l'algorithme de base, nous devons spécifier le type de retour correspondant pour chaque méthode *accept* ce qui nous donne deux méthodes *accept* : Integer accept(PerimeterVisitor v) et String accept(ToStringVisitor v). Les types de retour associés aux méthodes *accept* seront identifiés à partir des méthodes *visit* définies dans la partie Visiteur.

[1] http://plugins.jetbrains.com/plugin/?idea_ce&id=6889

I.B ForAll v in $\mathbb{V}$ do **AddSpecializedMethodWithGenerics**(S,"accept",$\mathbb{R}$,"Visitor",v)	(remplace I)

Algorithme 10: Algorithme de la transformation du Visiteur vers le Composite de la variation types génériques

Pour traiter cette variation, nous appliquons l'opération *AddSpecialisedMethodWithGenerics* de l'étape I.B.

Précondition générée et interprétations

L'intégration des nouvelles opérations impliquées dans cette transformation dans le calcul des préconditions a généré la précondition 2 qui garantisse la réussite de la transformation réversible de cette variation.

La différence signifiante de ces préconditions par rapport à celles de la transformation de base est la vérification des types de retour qui ne doivent pas être primitifs. Ceci est manifesté par $\neg$`IsPrimitiveType`$(String)$ et $\neg$`IsPrimitiveType`$(Integer)$. Cette précondition est nécessaire pour exécuter l'opération *PullUpWithGenerics*. En fait, les types génériques sont adaptés uniquement aux types classes et non pas aux types primitifs ce qui explique l'existence de cette précondition.

$\neg$ExistsMethodInvocation($Ellipse, toStringAux, Ellipse, perimeter$)
$\wedge \neg$ExistsMethodDefinition($Graphic, accept$)
$\wedge \neg$ExistsMethodDefinition($Ellipse, accept$)
$\wedge \neg$ExistsMethodDefinition($Composite, accept$)
$\wedge \neg$IsInheritedMethod($Graphic, accept$)
$\wedge \neg$IsUsedMethodIn($Graphic, toStringAux, Ellipse$)
$\wedge \neg$IsUsedMethodIn($Graphic, toStringAux, Composite$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($Composite, toString, this$)
$\wedge$ AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC($Ellipse, toString, this, Graphic$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($Ellipse, toString, this$)
$\wedge \neg$IsPrimitiveType($String$)
$\wedge \neg$ExistsAbstractMethod($Visitor, visit$)
$\wedge \neg$IsPrimitiveType($Integer$)
$\wedge \neg$ExistsType($Visitor$)
$\wedge$ ExistsClass($Ellipse$)
$\wedge \neg$BoundVariableInMethodBody($Graphic, toString, v$)
$\wedge \neg$BoundVariableInMethodBody($Graphic, perimeter, v$)
$\wedge$ IsRecursiveMethod($Composite, toString$)
$\wedge$ ExistsClass($Composite$)
$\wedge$ IsRecursiveMethod($Composite, perimeter$)
$\wedge$ ExistsMethodDefinitionWithParams($Graphic, toString, [\,]$)
$\wedge$ ExistsAbstractMethod($Graphic, toString$)
$\wedge \neg$IsInheritedMethod($Graphic, toStringAux$)
$\wedge \neg$IsInheritedMethodWithParams($Graphic, toStringAux, [\,]$)
$\wedge \neg$ExistsMethodDefinitionWithParams($Graphic, toStringAux, [\,]$)
$\wedge$ HasReturnType($Graphic, toString, String$)
$\wedge$ ExistsMethodDefinition($Graphic, toString$)
$\wedge$ ExistsMethodDefinition($Ellipse, toString$)
$\wedge$ ExistsMethodDefinition($Composite, toString$)
$\wedge \neg$ExistsMethodDefinition($Graphic, toStringAux$)
$\wedge \neg$ExistsMethodDefinition($Ellipse, toStringAux$)
$\wedge \neg$ExistsMethodDefinition($Composite, toStringAux$)
$\wedge$ ExistsClass($Graphic$)
$\wedge$ IsAbstractClass($Graphic$)
$\wedge$ ExistsMethodDefinitionWithParams($Graphic, perimeter, [\,]$)
$\wedge$ ExistsAbstractMethod($Graphic, perimeter$)
$\wedge \neg$IsInheritedMethod($Graphic, perimeterAux$)
$\wedge \neg$IsInheritedMethodWithParams($Graphic, perimeterAux, [\,]$)
$\wedge \neg$ExistsMethodDefinitionWithParams($Graphic, perimeterAux, [\,]$)
$\wedge$ AllSubclasses($Graphic, [Ellipse; Composite]$)
$\wedge$ HasReturnType($Graphic, perimeter, Integer$)
$\wedge \neg$IsPrivate($Graphic, perimeter$)
$\wedge \neg$IsPrivate($Ellipse, perimeter$)
$\wedge \neg$IsPrivate($Composite, perimeter$)
$\wedge$ ExistsMethodDefinition($Graphic, perimeter$)
$\wedge$ ExistsMethodDefinition($Ellipse, perimeter$)
$\wedge$ ExistsMethodDefinition($Composite, perimeter$)
$\wedge \neg$ExistsMethodDefinition($Graphic, perimeterAux$)
$\wedge \neg$ExistsMethodDefinition($Ellipse, perimeterAux$)
$\wedge \neg$ExistsMethodDefinition($Composite, perimeterAux$)
$\wedge \neg$ExistsType($ToStringVisitor$)
$\wedge \neg$ExistsType($PerimeterVisitor$)

Precondition 2: Préconditions garantissant le non-échec de la transformation réversible de la variation méthodes avec types de retour

3 Plusieurs niveaux hiérarchiques

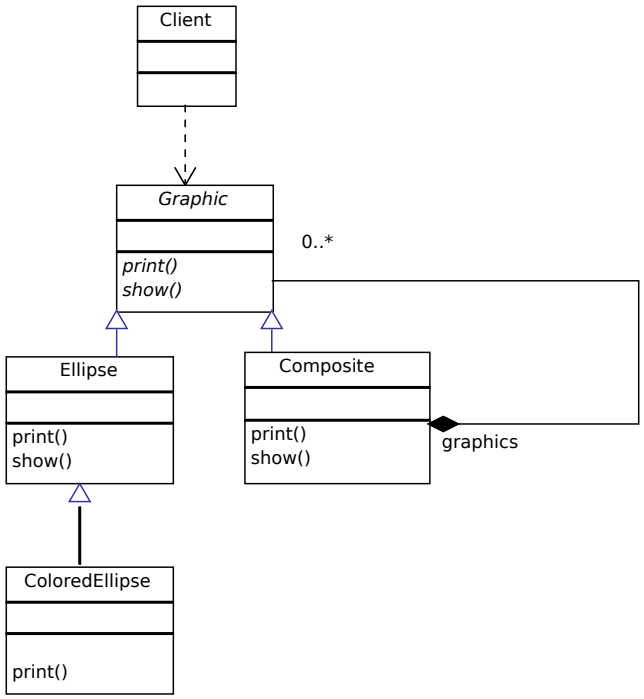
Structure Composite

Nous supposons que dans cette variation, le patron Composite comporte plusieurs niveaux hiérarchiques avec une répartition aléatoire des définitions des méthodes métiers sur les classes composites. Dans le programme de la figure 5.2, nous remarquons que la classe Ellipse possède une sous classe dans laquelle la méthode print est redéfinie et l'autre méthode show est héritée.

Structure Visiteur

Pour avoir un Visiteur, on doit fournir une méthode *visit* pour chaque classe Composite.

Le code de la méthode show définie dans la classe *Ellipse* et héritée par la classe ColoredEllipse, est placée dans les méthodes visit(ColoredEllipse c) et visit(Ellipse e) dans la classe *PrettyPrintvisitor* (voir programme 14).



(a) Diagramme de classe.

```
abstract class Graphic {
  abstract public void print();
  abstract public void show();
}
```

```
class Ellipse extends Graphic{

 public void print() {
   System.out.println("Ellipse :" + this);
 }
 public void show(){
   System.out.println("Ellipse corresponding to the object " + this + ".");
 }
}
```

```
class ColoredEllipse extends Ellipse{
    int color;
    public void print() {
        System.out.println("Ellipse :" + color);
    }
}
```

```
class Composite extends Graphic {
 List<Graphic> graphics = new ArrayList<Graphic>();
 public void print() {
   System.out.println("Composite:");
   for (Graphic g : graphics) {
            g.print();
   }
 }
 public void show(){
  System.out.println("Composite " + this + " composed of:");
  for (Graphic g : graphics) {
      g.show();
  }
  System.out.println("(end of composite)");
 }

}
```

(b) Code source Java.

FIGURE 5.2: Un Composite avec plusieurs niveaux hiérarchiques.

```
abstract class Graphic {
    public void print() {
        accept(new PrintVisitor());
    }
    public void show() {
        accept(new ShowVisitor());
    }
    public abstract void accept(Visitor v);
}
```

```
class Ellipse extends Graphic{

    public void accept(Visitor v) {
        v.visit(this);
    }
}
```

```
class ColoredEllipse extends Ellipse{

    int color;

    public void accept(Visitor v) {
        v.visit(this);
    }
}
```

```
class Composite extends Graphic {
    List<Graphic> graphics = new ArrayList<Graphic>();

    public void accept(Visitor v) {
        v.visit(this);
    }
}
```

```
public class PrintVisitor extends Visitor {

    public void visit(Composite c) {
        System.out.println("Composite:");
        for (Graphic g : c.graphics) {
            g.accept(this);
        }
    }
    public void visit(Ellipse e) {
        System.out.println("Ellipse :" + e);
    }
    public void visit(ColoredEllipse col) {
        System.out.println("Ellipse :" + col.color);
    }

}
```

```
public class ShowVisitor extends Visitor {
    public void visit(Composite c) {
        System.out.println("Composite " + c + " composed of:");
        for (Graphic g : c.graphics) {
            g.accept(this);
        }
        System.out.println("(end of composite)");
    }
    public void visit(Ellipse e) {
        System.out.println("Ellipse corresponding to the object "
                                                    + e + ".");
    }
    public void visit(ColoredEllipse col) {
        System.out.println("Ellipse corresponding to the object "
                                                + col + ".");
    }
}
```

Programme 14: Un Visiteur avec plusieurs niveaux hiérarchiques

Transformation du Composite vers le Visiteur

Nous utilisons les notations suivantes dans l'algorithme 11 correspondant à cette variation :

Notation	Définition
$i(c)$	une fonction qui donne la liste des méthodes héritées d'une classe, ici, $i(\text{ColoredEllipse}) = \{\text{show()}\}$
$s(c)$	une fonction qui donne la classe mère d'une classe donnée.

1.C ForAll c in $\mathbb{C}$, ForAll m in $i(c)$ do **PushDownCopy**(c,m,$s(c)$) (avant 1)

Algorithme 11: Algorithme de la transformation du Composite vers le Visiteur de la variation à plusieurs niveaux hiérarchiques

Pour partir d'un Composite dans lequel toutes les méthodes métier sont définies dans toutes les composites et non pas héritées, nous devons copier chaque méthode héritée et le placer dans la classe qu'elle hérite (voir programme 15). Nous appliquons ainsi l'opération *PushDownCopy*. Cette opération consiste à appliquer les opérations *Extract Method*, *InlineMethod*, *Push Down Method* et *Rename*. Nous appliquons cette opération avant d'appliquer l'algorithme de base et nous appelons cette étape 1.C.

Transformation du Visiteur vers le Composite

Nous appliquons tout d'abord l'algorithme de base. Puis, afin de récupérer la structure initiale, nous devons supprimer les méthodes qui étaient héritées dans la structure initiale. Ceci est réalisé par l'application de l'opération *DeleteDuplicateMethod* dans l'étape XII.C. Les méthodes à supprimer étaient copiées dans les classes qui les héritent afin d'éviter la répartition aléatoire des méthodes métiers (étape 1.C de l'algorithme Composite vers Visiteur). L'algorithme 12 montre les différentes adaptations par rapport à l'algorithme de base.

Remarque L'opération de refactoring *DeleteMethod* doit vérifier que le code à supprimer n'a pas été déviée par rapport au code hérité. Dans le pratique, une telle méthode, qui sert à faire un *Pull Up Method*, n'est pas implémentée dans les outils de refactoring. Nous utilisons à sa place l'opération *Safe Delete* qui est plus sûre.

Précondition générée et interprétations

Les nouvelles opérations impliquées dans la transformation de cette variation ont été intégrées dans notre calcul de préconditions générant ainsi la précondition 3 qui vérifie le non-échec de la transformation de cette variation.

Nous avons deux préconditions intéressantes qui caractérisent cette variation :

- La méthode show doit être héritée par ColoredEllipse. Ceci se manifeste par l'apparition de la précondition `IsInheritedMethod`$(ColoredEllipse, show)$. Cette précondition est demandée par l'opération *PushDownCopy* puisque elle va vérifier que la méthode en question est héritée ou non.
- La méthode show de la classe ColoredEllipse ne doit appeler aucune méthode surchargée appelée avec l'argument this dans la classe Ellipse. Ceci est due au fait que, si la méthode Ellipse : :show appelle n'importe quelle méthode surchargée avec l'argument this, nous ne pouvons pas dupliquer cette méthode dans la classe ColoredEllipse car l'argument this va référer à cette classe, ce qui peut changer la sémantique de cette méthode qui est sensée garder la même sémantique dans les deux classes.

```
abstract class Graphic {
  abstract public void print();
  abstract public void show();
}
```

```
class Ellipse extends Graphic{

 public void print() {
   System.out.println("Ellipse :" + this);
 }
 public void show(){
   System.out.println("Ellipse corresponding to the object " + this + ".");
 }
}
```

```
class ColoredEllipse extends Ellipse{
    int color;
    public void print() {
        System.out.println("Ellipse :" + color);
    }

    public void show(){
   System.out.println("Ellipse corresponding to the object " + this + ".");
 }
}
```

```
class Composite extends Graphic {

 List<Graphic> graphics = new ArrayList<Graphic>();

 public void print() {
   System.out.println("Composite:");
   for (Graphic g : graphics) {
            g.print();
   }
 }
 public void show(){
  System.out.println("Composite " + this + " composed of:");
  for (Graphic g : graphics) {
      g.show();
  }
  System.out.println("(end of composite)");
 }

}
```

Programme 15: État d'un Composite avec plusieurs niveaux hiérarchiques sans répartition aléatoire des méthodes métiers.

XII.C ForAll (c,m) in $\mathbb{C}$, ForAll m in $i(c)$ do **DeleteDuplicateMethod**(c,m) (après XII)

Algorithme 12: Algorithme de la transformation du Visiteur vers le Composite de la variation à plusieurs niveaux hiérarchiques

$\texttt{IsInheritedMethod}(ColoredEllipse, show)$
$\wedge \neg\texttt{ExistsMethodInvocation}(Ellipse, showTmpVC, Ellipse, print)$
$\wedge \neg\texttt{ExistsMethodDefinition}(Graphic, accept)$
$\wedge \neg\texttt{ExistsMethodDefinition}(Ellipse, accept)$
$\wedge \neg\texttt{ExistsMethodDefinition}(ColoredEllipse, accept)$
$\wedge \neg\texttt{ExistsMethodDefinition}(Composite, accept)$
$\wedge \neg\texttt{IsInheritedMethod}(Graphic, accept)$
$\wedge \neg\texttt{IsUsedMethodIn}(Graphic, showTmpVC, Ellipse)$
$\wedge \neg\texttt{IsUsedMethodIn}(Graphic, showTmpVC, ColoredEllipse)$
$\wedge \neg\texttt{IsUsedMethodIn}(Graphic, showTmpVC, Composite)$
$\wedge \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(Composite, show, this)$
$\wedge \texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(Ellipse, show, this, Graphic)$
$\wedge \neg\texttt{ExistsType}(Visitor)$
$\wedge \texttt{ExistsClass}(Ellipse)$
$\wedge \neg\texttt{BoundVariableInMethodBody}(Graphic, show, v)$
$\wedge \neg\texttt{BoundVariableInMethodBody}(Graphic, print, v)$
$\wedge \texttt{IsRecursiveMethod}(Composite, show)$
$\wedge \texttt{ExistsClass}(Composite)$
$\wedge \texttt{IsRecursiveMethod}(Composite, print)$
$\wedge \texttt{ExistsMethodDefinitionWithParams}(Graphic, show, [\,])$
$\wedge \texttt{ExistsAbstractMethod}(Graphic, show)$
$\wedge \neg\texttt{IsInheritedMethod}(Graphic, showTmpVC)$
$\wedge \neg\texttt{IsInheritedMethodWithParams}(Graphic, showTmpVC, [\,])$
$\wedge \neg\texttt{ExistsMethodDefinitionWithParams}(Graphic, showTmpVC, [\,])$
$\wedge \texttt{HasReturnType}(Graphic, show, void)$
$\wedge \texttt{ExistsMethodDefinition}(Graphic, show)$
$\wedge \texttt{ExistsMethodDefinition}(Composite, show)$
$\wedge \neg\texttt{ExistsMethodDefinition}(Graphic, showTmpVC)$
$\wedge \neg\texttt{ExistsMethodDefinition}(Ellipse, showTmpVC)$
$\wedge \neg\texttt{ExistsMethodDefinition}(ColoredEllipse, showTmpVC)$
$\wedge \neg\texttt{ExistsMethodDefinition}(Composite, showTmpVC)$
$\wedge \texttt{ExistsClass}(Graphic)$
$\wedge \texttt{IsAbstractClass}(Graphic)$
$\wedge \texttt{ExistsMethodDefinitionWithParams}(Graphic, print, [\,])$
$\wedge \texttt{ExistsAbstractMethod}(Graphic, print)$
$\wedge \neg\texttt{IsInheritedMethod}(Graphic, printTmpVC)$
$\wedge \neg\texttt{IsInheritedMethodWithParams}(Graphic, printTmpVC, [\,])$
$\wedge \neg\texttt{ExistsMethodDefinitionWithParams}(Graphic, printTmpVC, [\,])$
$\wedge \texttt{AllSubclasses}(Graphic, [Ellipse; ColoredEllipse; Composite])$
$\wedge \texttt{HasReturnType}(Graphic, print, void)$
$\wedge \neg\texttt{IsPrivate}(Graphic, print)$
$\wedge \neg\texttt{IsPrivate}(Ellipse, print)$
$\wedge \neg\texttt{IsPrivate}(ColoredEllipse, print)$
$\wedge \neg\texttt{IsPrivate}(Composite, print)$
$\wedge \texttt{ExistsMethodDefinition}(Graphic, print)$
$\wedge \texttt{ExistsMethodDefinition}(Ellipse, print)$
$\wedge \texttt{ExistsMethodDefinition}(ColoredEllipse, print)$
$\wedge \texttt{ExistsMethodDefinition}(Composite, print)$
$\wedge \neg\texttt{ExistsMethodDefinition}(Graphic, printTmpVC)$
$\wedge \neg\texttt{ExistsMethodDefinition}(Ellipse, printTmpVC)$
$\wedge \neg\texttt{ExistsMethodDefinition}(ColoredEllipse, printTmpVC)$
$\wedge \neg\texttt{ExistsMethodDefinition}(Composite, printTmpVC)$
$\wedge \neg\texttt{ExistsType}(ShowVisitor)$
$\wedge \neg\texttt{ExistsType}(PrintVisitor)$
$\wedge \texttt{ExistsType}(ColoredEllipse)$
$\wedge \texttt{ExistsClass}(ColoredEllipse)$
$\wedge \texttt{IsSubType}(ColoredEllipse, Ellipse)$
$\wedge \neg\texttt{ExistsMethodDefinition}(ColoredEllipse, show)$
$\wedge \texttt{ExistsMethodDefinition}(Ellipse, show)$
$\wedge \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(Ellipse, show, this)$

Precondition 3: Préconditions garantissant le non-échec de la transformation réversible de la variation plusieurs niveaux hiérarchiques

4 Composite avec Interface au lieu d'une classe abstraite

Structure Composite

Dans cette variation, nous supposons que la racine du Composite est une interface et non pas une classe abstraite. Nous considérons qu'il y a une classe intermédiaire entre l'interface et ses sous classes. Nous voudrons simuler les mêmes structures de cette variation qui existent déjà dans des logiciels réels comme IntelliJIdea, JHotDraw...etc. Cette variation correspond au patron de conception *Facade* : nous supposons qu'il n'y pas d'autres classes qui implémentent l'interface à part la classe intermédiaire (voir programme 16).

```
interface Graphic {
  public void print();
  public void show();
}
```

```
public abstract class AbstractGraphic  implements Graphic{
  abstract public void print();
  abstract public void show();
}
```

```
class Ellipse extends AbstractGraphic {
  public void print() {
   System.out.println("Ellipse :" + this);
  }
  public void show(){
   System.out.println("Ellipse corresponding to the object " + this + ".");
  }
}
```

```
class Composite extends AbstractGraphic  {

    List<Graphic> graphics = new ArrayList<Graphic>();
    public void print() {
        System.out.println("Composite:");
        for (Graphic g : graphics) {
            g.print();
        }
    }
    public void show(){
        System.out.println("Composite " + this + " composed of:");
        for (Graphic g : graphics) {
            g.show();
        }
        System.out.println("(end of composite)");
    }

}
```

Programme 16: Composite avec Interface

Structure Visiteur

Le programme 17 montre la structure que nous voudrons produire à partir de la structure initiale.

Notez bien que l'objet g, qui reçoit le message de l'appel de la méthode *accept* à l'intérieure de la méthode *visit*, est de type Graphic et non pas AbstractGraphic. Cette information sera utile plus tard lorsque nous expliquons l'utilisation pratique de la transformation de cette variation.

Transformation du Composite vers le Visiteur

Pour obtenir la structure cible, nous devons créer des délégateurs print(){printaux(..)} et show(){showaux(..)} dans la classe AbstractGraphic, ensuite nous faisons un inline des appels récursifs de print et show dans la classe Composite (étapes 2 et 3). Mais les appels récursifs correspondent aux méthodes print() et show()

```
interface Graphic {
  public void print();
  public void show();

  public void accept(Visitor v);
}
```

```
public abstract class AbstractGraphic  implements Graphic{

  public void print(){
   accept(new Printvisitor());
  }
  public void show(){
   accept(new ShowVisitor());
  }
}
```

```
class Ellipse extends AbstractGraphic {

  public void accept(Visitor v) {
    v.visit(this);
  }
}
```

```
class Composite extends AbstractGraphic {
   List<Graphic> graphics = new ArrayList<Graphic>();

   public void accept(Visitor v) {
      v.visit(this);
  }


}
```

```
public class Printvisitor extends Visitor{

  public void visit(Composite c) {
   System.out.println("Composite:");
   for (Graphic g : c.graphics) {
     g.accept(this);
   }
  }
  public void visit(Ellipse e) {
   System.out.println("Ellipse :" + e);
  }
}
```

```
public class ShowVisitor extends  Visitor{

  public void visit(Composite c) {
    System.out.println("Composite " + this + " composed of:");
    for (Graphic g : c.graphics) {
      g.accept(this);
    }
    System.out.println("(end of composite)");
  }
  public void visit(Ellipse e) {
    System.out.println("Ellipse corresponding to the object " + e + ".");
  }
}
```

Programme 17: Visiteur avec Interface

déclarées dans l'interface Graphic, or les délégateurs sont définis dans la classe abstraite AbstractGraphic. Pour résoudre ce problème, nous introduisons un *downcast* vers la classe AbstractGraphic au niveau des appels récursifs de print et show comme suit : ((AbstractFigure) g).print() et ((AbstractFigure) g).show() (step 3.D). Ceci est nécessaire pour l'opération *inline* de l'outil de refactoring car cette opération ne s'applique pas aux méthodes abstraites. Ce *downcast* est légal parce que comme indiqué au-dessus que nous supposons qu'il n'y a pas des sous-classe qui implémentent l'interface à part la classe intermédiaire.

Après la création de la méthode *accept* (step 8), nous faisons monter (*pull up*) sa déclaration de la classe intermédiaire AbstractGraphic vers l'interface Graphic, puis nous supprimons le *downcast* car l'appel récursif vers la méthode *accept* est légal sans *downcast* vue que cette méthode est déclarée au niveau de l'interface Graphic qui est le type de l'objet qui l'invoque dans les classe Visiteurs (step 8.D).

L'algorithme 13 montre les différentes adaptations que nous devons faire par rapport à l'algorithme de base pour transformer cette variation de Composite.

3.D	ForAll m in $\mathbb{M}$, c in $\mathbb{C}$ do **IntroduceDownCast**(c,m,S)	(avant 3)
8.D	**pullupAbstractMethod**(S, "accept", I)	
	ForAll v in $\mathbb{V}$ do **DeleteDownCast**(v,"accept")	(après 8)

Algorithme 13: Algorithme de la transformation du Composite vers le Visiteur de la variation interface

Le cas pratique de cette transformation. L'algorithme présenté ci-dessus montre la solution idéale pour obtenir la bonne structure du Visiteur. En fait, il n'existe pas une opération de refactoring dans les outils de refactoring qui permet de faire un *downcast*. Pour faire une transformation totalement automatique, nous n'utilisons pas les *downcast* ainsi que l'application de *inline* au niveau des appels des délégateurs. Ainsi nous obtenons le programme 18.

Nous remarquons que chaque appel récursif va engendrer la création d'une nouvelle instance du Visiteur en question. Le résultat ne viole pas le rôle d'un Visiteur mais il influence sur la performance de la mémoire. Mais ce problème d'optimisation va disparaitre un fois qu'on retourne vers la structure initiale. Pour obtenir la structure idéale qui correspond à un Composite avec interface il sera intéressant d'envisager l'implémentation d'une opération de refactoring qui traite les *downcast*. Donc, dans la vie pratique, nous appliquons l'algorithme de base sauf l'étape 3.

Transformation du Composite vers le Visiteur

Après l'application de la solution pratique de la transformation Composite→Visiteur, nous appliquons l'algorithme de base sans passer par l'étape VII car son exécution dépend de l'existence de l'appel récursif vers la méthode *accept*, ce qui n'est pas le cas ici.

Si nous appliquons la solution idéale de la transformation Composite→Visiteur (avec le *downcast*), après l'application de l'étape VII, nous pouvons supprimer les *downcasts*.

Précondition générée et interprétations

Nous considérons uniquement la solution pratique dans la génération des préconditions. La précondition 4 est générée après la prise en compte de la solution réelle et qui garantit le non-échec de la transformation réversible de cette variation dans le cas pratique (sans *downcast*) :

Vu que ne traitons pas la solution idéale ici, nous n'appliquons pas l'étape 3, ce qui explique l'absence de la précondition `IsRecursiveMethod(Group, show)` de l'ensemble des préconditions. Ce prédicat est demandée essentiellement par l'opération *InlineMethodInvocation* qui vérifie que l'appel de la méthode en question est récursif.

```
interface Graphic {
  public void print();
  public void show();
}
```

```
public abstract class AbstractGraphic  implements Graphic{
  public void print(){
     accept(new Printvisitor());
  }
  public void show(){
     accept(new ShowVisitor());
  }
  abstract void accept(Visitor v);
}
```

```
class Ellipse extends AbstractGraphic {

  public void accept(Visitor v) {
     v.visit(this);
  }
}
```

```
class Composite extends AbstractGraphic {
  List<Graphic> graphics = new ArrayList<Graphic>();

  public void accept(Visitor v) {
     v.visit(this);
  }
}
```

```
public class Printvisitor extends Visitor{

 public void visit(Composite c) {
  System.out.println("Composite:");
  for (Graphic g : c.graphics) {
     g.print();
  }
 }
 public void visit(Ellipse e) {
  System.out.println("Ellipse :" + e);
 }
}
```

```
public class ShowVisitor extends  Visitor{

    public void visit(Composite c) {
        System.out.println("Composite " + this + " composed of:");
        for (Graphic g : c.graphics) {
            g.prettyprint();
        }
        System.out.println("(end of composite)");
    }

    public void visit(Ellipse e) {
        System.out.println("Ellipse corresponding to the object " + e + ".");
    }

}
```

Programme 18: Visiteur avec Interface obtenu par la solution pratique

$\neg$ExistsMethodDefinition($Ellipse, accept$)
$\wedge \neg$ExistsMethodDefinition($Composite, accept$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($Composite, show, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($Ellipse, show, this$)
$\wedge \neg$ExistsType($Visitor$)
$\wedge$ IsRecursiveMethod($Composite, show$)
$\wedge$ IsRecursiveMethod($Composite, print$)
$\wedge$ HasReturnType($Ellipse, show, void$)
$\wedge$ HasReturnType($Composite, show, void$)
$\wedge$ ExistsMethodDefinition($Ellipse, show$)
$\wedge$ ExistsMethodDefinition($Composite, show$)
$\wedge \neg$ExistsMethodDefinition($Ellipse, showTmpVC$)
$\wedge \neg$ExistsMethodDefinition($Composite, showTmpVC$)
$\wedge$ HasReturnType($Ellipse, print, void$)
$\wedge$ HasReturnType($Composite, print, void$)
$\wedge \neg$IsPrivate($Ellipse, print$)
$\wedge \neg$IsPrivate($Composite, print$)
$\wedge$ ExistsMethodDefinition($Ellipse, print$)
$\wedge$ ExistsMethodDefinition($Composite, print$)
$\wedge \neg$ExistsMethodDefinition($Ellipse, printTmpVC$)
$\wedge \neg$ExistsMethodDefinition($Composite, printTmpVC$)
$\wedge \neg$ExistsType($ShowVisitor$)
$\wedge \neg$ExistsType($PrintVisitor$)
$\wedge \neg$ExistsType($AbstractGraphic$)
$\wedge$ IsInterface($Graphic$)
$\wedge$ ExistsType($Graphic$)
$\wedge$ ExistsClass($Ellipse$)
$\wedge$ ExistsClass($Composite$)
$\wedge$ ExtendsDirectly($Ellipse, java.lang.Object$)
$\wedge$ ExtendsDirectly($Composite, java.lang.Object$)
$\wedge$ implementsDirectly($Ellipse, Graphic$)
$\wedge$ implementsDirectly($Composite, Graphic$))

Precondition 4: Préconditions garantissant le non-échec de la transformation réversible de la variation interface dans le cas pratique

5 Bilan

Nous avons vu dans ce chapitre que les variations au niveau de la structure du patron Composite implique d'une part le changement des algorithmes de transformation de base, et d'autre part l'occurrence des variations au niveau de la structure Visiteur correspondante (par exemple le Visiteur avec types génériques). Nous avons traité jusqu'à maintenant quatre variations au niveau des patrons de conception Composite et Visiteur, mais nous ne pouvons pas évaluer le taux de ces variations dans le nombre total des variations vu que selon notre connaissance il n'y pas de travaux qui classifient toutes les variations de ces patrons. Nous allons voir que ces variations ont été identifiées dans une étude de cas réel JHotDraw que nous traitons dans le chapitre suivant.

6

Validation par étude de cas réel

Sommaire

Dans ce chapitre, nous validons notre approche sur une étude de cas réel JHotDraw [GI]. La validation comprend l'application de la transformation "Composite → Visiteur ; Visiteur → Composite". Elle comprend aussi le calcul des préconditions qui garantissent le bon déroulement de cette transformation, ainsi qu'un scénario de maintenance qui illustre l'utilité de notre transformation.

1 L'Étude de cas JHotDraw

JHotDraw a été le sujet d'étude dans quelques travaux du monde de la modularité. Ceccato et al. [CMM+05] utilise JHotDraw comme un logiciel qui contient des préoccupations dispersées dans des modules différents et qui nécessitent une analyse pour localiser les modules touchés par ces préoccupations et ensuite les préparer pour être traités par la programmation orienté aspect. Canfora and Cerulo [CC05] étudient aussi l'évolution de quelques préoccupations dans JHotDraw pour faciliter l'identification des aspects qui leurs correspondent.

Dans notre approche nous utilisons JHotDraw comme un programme qui contient une instance d'un patron Composite qui est caractérisé par l'existence des quatre variations que nous avons étudié dans le chapitre 5.

Détection du patron Composite

Nous utilisons un outil de détection de patrons qui s'appelle *pattern4* [TCSH06] pour détecter les patrons existent dans le programme JHotDraw. Nous avons détecté deux instances du patron Composite, une qui est très primitive (deux classes seulement) et une autre plus sophistiquée qui totalise 18 classes et 6 méthodes métier et qui rassemble toutes les variations que nous avons étudiées dans le chapitre 5. Nous choisissons cette instance et nous montrons dans la section suivante les quatre variations du Composite qui y existent.

L'instance du patron Composite sur laquelle nous appliquons la transformation est représentée par le diagramme 1.

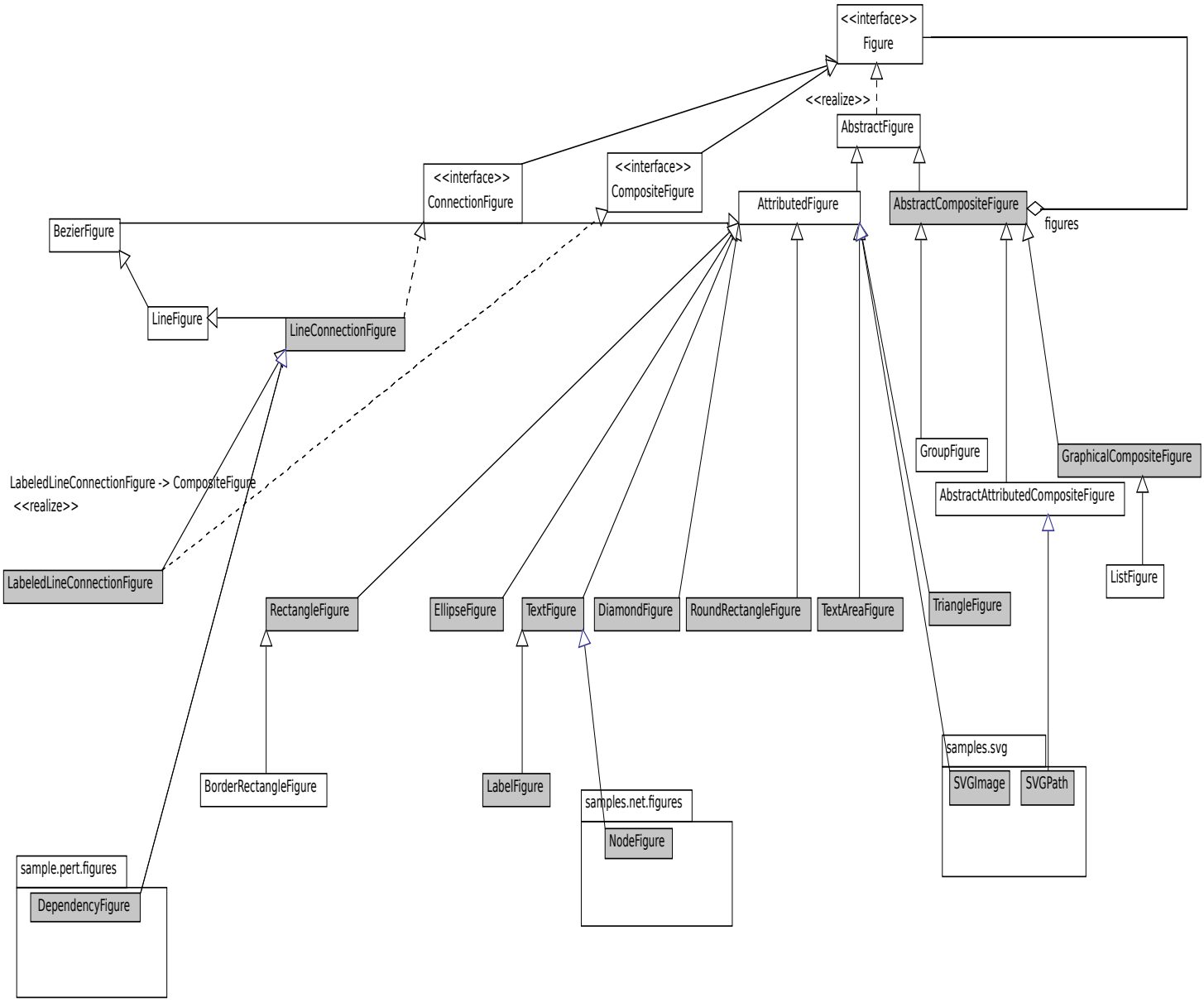


Diagramme 1: Classes intervenant dans la structure Composite de JHotDraw.

Variations du patron Composite dans JHotDraw

Nous avons identifié les quatre variations du patron Composite étudiées dans le chapitre 5. Nous donnons les parties clé du code qui représentent la trace de chaque variation.

La variation "méthodes avec paramètres". Cette variation est exprimée par la présence des paramètres au niveau des méthodes métier de l'instance du Composite que nous considérons dans JHotDraw. Toutes les méthodes métier dans cette étude de cas comportent des paramètres. Parmi ces méthodes il y en a celles qui ont plus qu'un paramètre. Le code ci-dessous donne une idée sur la signature des méthodes métier.

```
public interface Figure extends Cloneable, Serializable, DOMStorable {
  ...
  public void basicTransform(AffineTransform tx);

  public void setAttribute(AttributeKey key, Object value);

  public Figure findFigureInside(Point2D.Double p);

  boolean contains(Point2D.Double p);

  public void addNotify(Drawing d);

  public void removeNotify(Drawing d);

  ...

}
```

La variation "méthodes avec différents types de retour". Dans le code source ci-dessus, nous remarquons que les méthodes métier ont des types de retour hétérogènes. La méthode findFigureInside retourne une valeur de type Figure, la méthode contains retourne une valeur de type booléen et les quatre autres méthodes n'ont pas de type de retour mais elles ont le modifier void. Nous avons vu dans la description de la variation méthodes avec différents types de retour du chapitre 5 que pour transformer un Composite de cette variation vers un Visiteur à types génériques, il faut que les types de retour des méthodes soient des types classes et non pas des types primitifs. Pour résoudre ce problème au niveau des méthodes métier de JHotDraw, nous recourrons au tâches suivantes :

- Transformer le type boolean vers Boolean. Pour automatiser cette tâche, nous avons défini une opération de refactoring qui se base sur l'opération *change method signature*.
- Transformer le *modifier* void vers le type Object. Nous devons ajouter une instruction return qui retourne la valeur nulle pour chaque méthode qui a subit cette tâche car la méthode doit retourner une valeur avec cette nouvelle signature. L'opération qui automatise cette tâche se base aussi sur l'opération de refactoring *change method signature* avec l'ajout de l'instruction indiquée. Noter que l'ajout de cette instruction ne fait pas parti de refactoring, mais nous faisons qu'intervenir directement au niveau de l'arbre syntaxique sous la condition que nous connaissons que cette tâche ne viole pas la sémantique du programme. Voici ci-dessous un code source qui montre l'état d'une méthode qui n'a pas initialement un type de retour et qui a subit cette tâche.

```
public class EllipseFigure extends AttributedFigure {
 ...
 public Object basicTransform(AffineTransform tx) {
  Point2D.Double anchor = getStartPoint();
  Point2D.Double lead = getEndPoint();
```

```
      basicSetBounds(
        (Point2D.Double) tx.transform(anchor, anchor),
        (Point2D.Double) tx.transform(lead, lead)
      );
      return null;
     }
```

La variation "hiérarchie avec plusieurs niveaux". L'instance du Composite que nous considérons dans JHotDraw montre plusieurs niveaux hiérarchiques avec des répartitions aléatoires des redéfinitions des méthodes métier. Par exemple dans le code source ci-dessous, la méthode findFigureInside est définie dans la classe AbstractCompositeFigure mais elle n'est pas redéfinie dans sa sous classe GraphicalCompositeFigure comme les autres méthodes. Ceci doit être pris en compte dans la transformation Composite ↔ Visiteur comme déjà expliqué dans le chapitre5.

```
public abstract class AbstractCompositeFigure extends AbstractFigure
                                                  implements CompositeFigure {
  public Figure findFigureInside(Point2D.Double p){
   ...
  }

  public void basicTransform(AffineTransform tx){
   ...
  }

  public void setAttribute(AttributeKey key, Object value){
   ...
  }

  boolean contains(Point2D.Double p){
   ...
  }

  public void addNotify(Drawing d){
   ...
  }

  public void removeNotify(Drawing d){
   ...
  }
}

public class GraphicalCompositeFigure extends AbstractCompositeFigure {

 public void basicTransform(AffineTransform tx){
   ...
  }

  public void setAttribute(AttributeKey key, Object value){
   ...
  }

  boolean contains(Point2D.Double p){
   ...
  }
```

```
  public void addNotify(Drawing d){
   ...
  }

  public void removeNotify(Drawing d){
   ...
  }
}
```

La variation interface. La variation interface qui caractérise l'instance du composite que nous traitons dans JHotDraw est représentée par l'interface Figure qui est la racine de cette instance du Composite et par la classe abstraite AbstractFigure qui joue le rôle d'une intermédiaire entre l'interface et les classes composites (voir le diagramme 1).

2 Transformation de JHotDraw

Nous appliquons une transformation réversible sur l'application de JHotDraw. Les données de cette transformation sont comme suit :

- $S =$ AbstractFigure.
- $\mathbb{C} =$ { EllipseFigure, DiamondFigure, RectangleFigure, RoundRectangleFigure, TriangleFigure, TextFigure, BezierFigure, TextAreaFigure,NodeFigure, SVGImage, SVGPath, DependencyFigure, LineConnectionFigure, LabeledLineConnectionFigure, AbstractCompositeFigure , GraphicalCompositeFigure }.
- $\mathbb{M}_{\mathbb{P}} =$ { basicTransform (AffineTransform tx), contains(Point2D.Double p), setAttribute(AttributeKey key,Object value), findFigureInside(Point2D.Double p), addNotify(Const "Drawing d),removeNotify(Drawing d)}.
- $\mathbb{M}_{\mathbb{W}} = \emptyset$.
- $\mathbb{R} =$ { (basicTransform,Void), (contains,Boolean), (setAttribute, Void), (findFigureInside,Figure), (addNotify, Void), (removeNotify, Void)}.
- i(LineConnectionFigure) = {findFigureInside, setAttribute,contains}
  i(SVGPath) = {findFigureInside, addNotify,removeNotify},
  i(LabeledLineConnectionFigure) = {basicTransform, setAttribute,findFigureInside, contains},
  i(DependencyFigure) = {addNotify, basicTransform,setAttribute, findFigureInside, contains},
  i(NodeFigure) = {addNotify, basicTransform,setAttribute, findFigureInside, contains},
  i(GraphicalCompositeFigure) = {findFigureInside}
- s(LineConnectionFigure)= {BezierFigure},
  s(SVGPath)= {AbstractCompositeFigure},
  s(LabeledLineConnectionFigure)= {BezierFigure},
  s(DependencyFigure)= {LineConnectionFigure},
  s(NodeFigure)= {TextFigure},
  s(GraphicalCompositeFigure)= {AbstractCompositeFigure}

Transformation du Composite vers le Visiteur

Nous présentons ici la structure cible de la transformation Composite $\rightarrow$ Visiteur, puis nous donnons l'algorithme de transformation.

Structure cible de JHotDraw. La structure JHotDraw que nous voudrons atteindre avec cette transformation est caractérisée par un patron Visiteur à types génériques et dont les grands traits sont montrés par le code source suivant :

```
public abstract class AbstractFigure implements Figure {

 public void addNotify(Drawing d) {
  accept(new AddNotifyVisitor(d));
  }
 public void removeNotify(Drawing d) {
  accept(new RemoveNotifyVisitor(d));
 }
 public void basicTransform(AffineTransform ty) {
  accept(new BasicTransformVisitor(ty));
 }
 public Figure findFigureInside(Point2D.Double p) {
  return accept(new FindFigureInsideVisitor(p));
 }
 public void setAttribute(AttributeKey name, Object value) {
  accept(new SetAttributeVisitor(name, value));
 }
 public boolean contains(Point2D.Double p) {
  return accept(new ContainsVisitor(p));
 }
 public abstract <T> T accept(Visitor<T> v);}
```

```
public abstract class Visitor<T> {

  public abstract T visit(RectangleFigure re);

  public abstract T visit(EllipseFigure el);

  public abstract T visit(TextFigure te);

  public abstract T visit(LineConnectionFigure li);

  public abstract T visit(LabeledLineConnectionFigure la);

  public abstract T visit(BezierFigure be);

  public abstract T visit(DiamondFigure di);

  public abstract T visit(GraphicalCompositeFigure gr);

  public abstract T visit(AbstractCompositeFigure ab);

  public abstract T visit(RoundRectangleFigure ro);

  public abstract T visit(TriangleFigure tr);

  public abstract T visit(TextAreaFigure te);

  public abstract T visit(SVGPath sv);

  public abstract T visit(SVGImage sv);

  public abstract T visit(NodeFigure no);}
```

```
public class BasicTransformVisitor extends Visitor<Object> {
  ...
  public Object visit(LabeledLineConnectionFigure la) {
        la.path.transform(getTy());
        la.invalidate();
        la.updateConnection();
        for (Figure f : la.children) {
            f.basicTransform(getTy());
        }
        la.invalidateBounds();
        return null;
    }
  public Object visit(RectangleFigure re) {...}
  ...
}
```

```
public class FindFigureInsideVisitor extends Visitor<Figure> {
 ...}
```

```
public class ContainsVisitor extends Visitor<Boolean> {
...}
```

```
public class SetAttributeVisitor extends Visitor<Object> {
...}
```

```
public class RemoveNotifyVisitor extends Visitor<Object> {
...}
```

```
public class AddNotifyVisitor extends Visitor<Object> {
...}
```

Cette structure est caractérisée par deux propriétés clés : les types génériques illustrés clairement dans le code et l'absence de l'appel récursif vers la méthode *accept* comme montré au niveau du code de la méthode visit(LabeledLineConnectionFigure la) de la classe BasicTransformVisitor. Ceci est du au choix de l'application pratique de la transformation de la variation interface et de la non utilisation du *down cast* (voir chapitre 5).

Algorithme de transformation. Pour obtenir la structure du Visiteur au sein de JHotDraw, nous appliquons la séquence des étapes suivante : 1.C ; 2 ; 4.A ; 5 ; 6.B ; 7 ; 8.

Les étapes avec des lettres an majuscules à la fin sont les étapes expliquées déjà dans la chapitre 5. Les autres étapes sont les étapes issues de l'algorithme de base présenté dans le chapitre 3 (page 44).

Transformation du Visiteur vers le Composite

La structure cible de cette transformation est la structure Composite de JHotDraw, nous présentons ici seulement l'enchainement des étapes qui permettent de récupérer cette structure (ces étapes sont déjà détaillées dans les chapitres 3 et 5) : I.B ; III ; IV ; V ; VI.A ; VIII ; IX ; X ; XI.A ; XI ; XII ; XII.C.

Le temps d'exécution de la transformation est de l'ordre de 16 minutes pour chaque sens de la transformation avec l'outil de refactoring offert avec IntelliJ IDEA (version 12.0.4).

Précondition générée

Nous avons généré la précondition qui garantit statiquement la réussite de la transformation réversible entre le patron Composite de JHotDraw et le Visiteur (voir annexe C). Le résultat montre que cette précondition vérifie l'existence et la bonne orchestration des variations que nous avons traité dans le chapitre 5.

3 Intérêt de la transformation de JHotDraw

Maintenances modulaires

L'instance du Composite que nous traitons dans ce chapitre contient 18 classes (y compris l'interface et la classe abstraite) et 6 méthodes métier. Comme la structure du Composite permet aux programme de se décomposer selon les types de données, nous pouvons ajouter facilement des nouvelles figures à JHotDraw. Ceci se fait par la création d'une sous classe de la classe AbstractFigure et l'implémentation des six méthodes métier. Il s'agit dans ce cas d'une tâche de maintenance modulaire.

Maintenant, nous voulons faire une évolution de l'une des méthodes métier au niveau de la structure Composite de JHotDraw, nous devons parcourir les 16 classes dans lesquelles cette méthode est définie et d'où nous devons faire des maintenances transversales qui sont coûteuses (voir la figure 6.1 qui visualise la dispersion des tâches de maintenance). Pour éviter ce type de maintenance transversale, nous passons à la structure Visiteur de JHotDraw qui est plus adaptée à ce type de maintenance. Au niveau de la structure Visiteur, nous faisons la tâche de maintenance indiquée dans un seul module comme visualisé dans la figure 6.2, puis nous revenons à la structure initiale de JHotDraw.

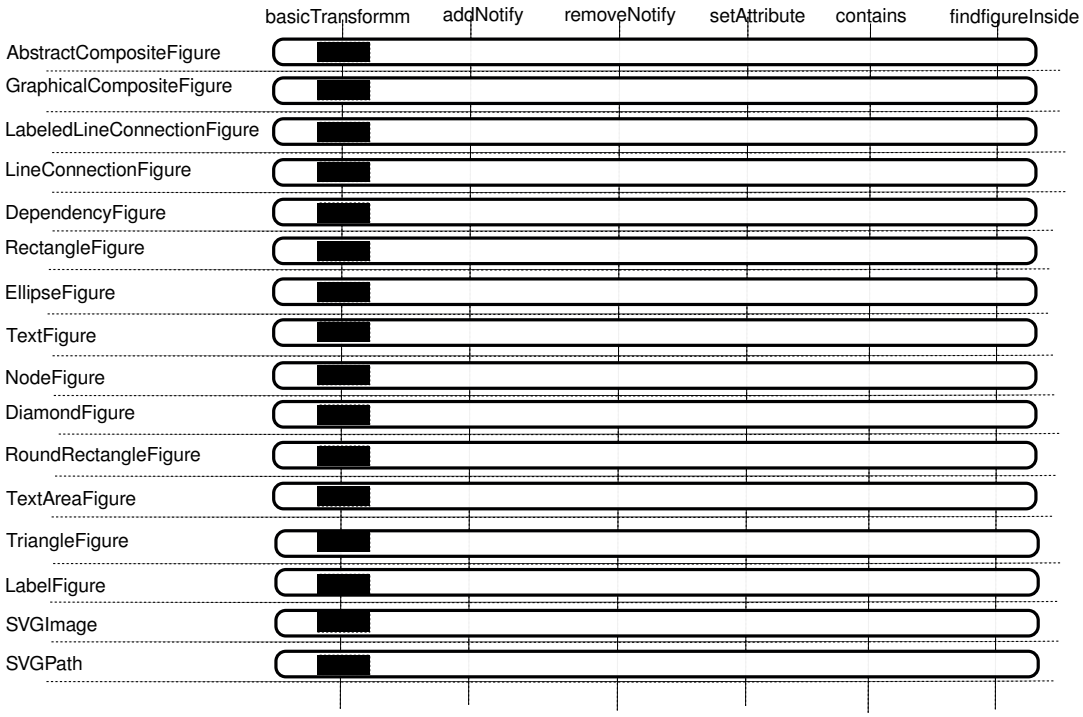


FIGURE 6.1: Maintenance Transversale dans la structure Composite de JHotDraw.

Scénario de maintenance

Nous appliquons ici un cas de maintenance réel sur la méthode métier basicTransform. Cette méthode est définie pour déplacer ou déformer les figures. Nous ajoutons à cette méthode une sorte de traçage pour afficher à l'utilisateur le type de figure qui est entrain d'être déplacé ou déformé et la référence de l'objet correspondant à cette figure. Si on applique cette tâche de maintenance sur la structure Composite de JHotDraw, la tâche de maintenance sera transverse vu qu'elle est de type fonctions et nous allons parcourir les 16 classes qui définissent cette méthode. Nous transformons alors la structure Composite de JHotDraw vers sa structure Visiteur pour pouvoir effectuer cette tâche de maintenance d'une façon modulaire.

Après la transformation du JHotDraw vers sa structure Visiteur, nous appliquons la tâche indiquée sur la classe BasicTransfomVisitor qui correspond au code métier de la méthode basicTransform. Voici ci dessous le code source qui corresponds à cette évolution :

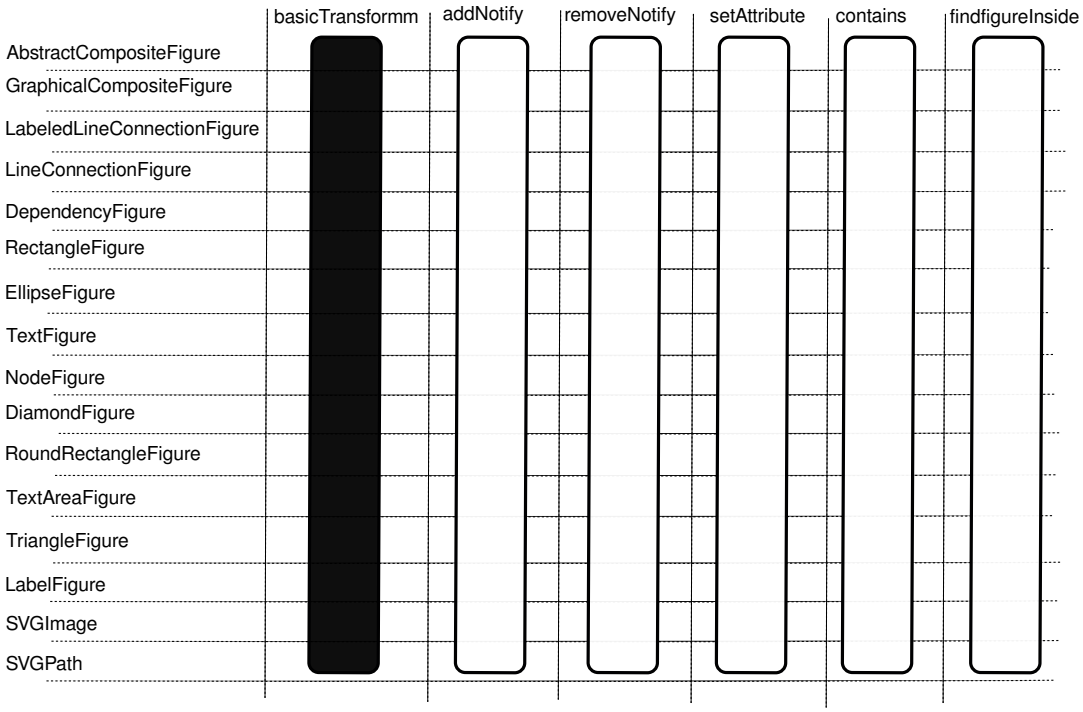


FIGURE 6.2: La façon modulaire de la maintenance montrée par la figure 6.1.

```
public class BasicTransformVisitor extends Visitor<Object> {
  ...

  public Object visit(RectangleFigure re) {
    ...
    System.out.println("Transforming the Rectangle...."+re);
    ...
  }
  public Object visit(EllipseFigure el) {
   ...
   System.out.println("Transforming the Ellipse...."+el);
   ...
  }
  public Object visit(LabeledLineConnectionFigure la) {
   ...

  }

 ...
}
```

L'affichage du console ci-dessous montre l'effet de la tâche de maintenance indiquée sur le comportement de la fonctionnalité basicTransform. Ceci s'affiche lorsque on dessine par exemple une ellipse ou un rectangle, puis on les déplace.

```
 ...
Transforming the Ellipse....EllipseFigure@18061339
Transforming the Ellipse....EllipseFigure@18061339
Transforming the Ellipse....EllipseFigure@18061339
Transforming the Ellipse....EllipseFigure@18061339
Transforming the Ellipse....EllipseFigure@18061339
Transforming the Ellipse....EllipseFigure@18061339
Transforming the Ellipse....EllipseFigure@18061339
Transforming number of the RoundRectangleFigure....
                                              RoundRectangleFigure@10656878
Transforming number of the RoundRectangleFigure....
                                              RoundRectangleFigure@10656878
Transforming the Rectangle....RectangleFigure@2978880
Transforming the Rectangle....RectangleFigure@2978880
...
```

Voici quelques échantillons du code qui est ajouté avec la tâche de maintenance évolutive modulaire effectuée dans la structure Visiteur de JHotDraw et qui est réparti sur les bons endroits par la transformation retour :

```
public class EllipseFigure extends AttributedFigure {
 ...
 public void basicTransform(AffineTransform ty) {
  Point2D.Double anchor = getStartPoint();
  Point2D.Double lead = getEndPoint();
  basicSetBounds(
   (Point2D.Double) ty.transform(anchor, anchor),
   (Point2D.Double) ty.transform(lead, lead)
  );
  System.out.println("Transforming the Ellipse...."+ this);
 }
 ...

}
```

```
public class EllipseFigure extends AttributedFigure {
 ...
 public void basicTransform(AffineTransform ty) {
   Point2D.Double anchor = getStartPoint();
   Point2D.Double lead = getEndPoint();
   basicSetBounds(
    (Point2D.Double) ty.transform(anchor, anchor),
    (Point2D.Double) ty.transform(lead, lead)
   );
   System.out.println("Transforming the Rectangle...."+ this);
 }
 ...
}
```

Dans toues les 16 classes qui définissent la méthode basicTransform, le code qui est ajouté au module BasicTransformVisitor de la structure Visiteur sera désormais injecté automatiquement dans les bons endroits. Nous remarquons que le nom de chaque objet qui référait dans la structure Visiteur aux composites est remplacée par une liaison statique this. Le code que nous avons ajouté dans la structure Visiteur est fonctionnel aussi dans la structure Composite.

Le scénario de maintenance que nous faisons ici est utilisé uniquement pour illustrer l'utilisation de la transformation dans les tâches de maintenances primitives. Une validation reste à faire dans ce contexte par l'élaboration d'une étude plus large sur le degré d'acceptabilité de notre transformation par les développeurs

dans leurs tâches de maintenances.

4 Bilan

Nous avons montré dans ce chapitre comment appliquer notre transformation sur le programme réel JHotDraw qui était le sujet de plusieurs travaux dans la littérature. Cette application comporte les quatre variations que nous avons traité dans le chapitre 5. Le rassemblement de ces quatre variations dans un seul programme a nécessité l'orchestration des différentes étapes des quatre algorithmes des variations. Cette orchestration est prouvée correcte à partir des résultats du calcul de la précondition qui vérifie que la transformation est seine et sauve. Par contre, nous avons encore quelques limites dans l'automatisation totale de la transformation vu que nous utilisons d'une façon très restreinte le changement de l'arbre syntaxique dans les situations que nous prévoyons que nous ne violons pas la sémantique du programme (comme par exemple l'ajout d'une instruction qui retourne une valeur nulle lorsque nous convertissons le modifier void par Object). Nous avons aussi montré dans ce chapitre comment notre transformation permet au développeur de faire sa tâche de maintenance d'une façon modulaire même si la structure de son programme initial ne permet pas.

7

Introduction/Suppression du patron Singleton

Sommaire

Dans les chapitres 3, 4 et 5, nous avons traité une transformation entre deux patrons de conception complémentaires de point de vue architecturale. Nous présentons dans ce chapitre une transformation au sein du patron Singleton où son introduction peut être bénéfique à l'optimisation et sa suppression peut être aussi intéressante quand on cherche plus de souplesse suite à de nouvelles exigences du programme. Ainsi, nous montrons comment réaliser cette transformation par la définition de deux algorithmes et nous présentons les nouvelles opérations qui sont utilisées dans ces algorithmes, puis nous discutons la précondition minimale de cette transformation.

1 Présentation du patron Singleton

Le patron Singleton tel qu'il est présenté par Gamma et al. [GHJV95] est un patron qui permet à une classe de ne s'instancier qu'une seule fois et d'y accéder avec un point d'accès global. Le programme 19 montre une instance d'un patron Singleton. Dans ce genre d'instance, nous trouvons :

Une variable de classe. La variable de classe singleton qui est de type Singleton et qui reçoit la valeur null. La valeur null est explicite dans ce programme, mais elle peut être implicite car c'est la valeur par défaut d'un objet. Le mot clé static est obligatoire.

Un Constructeur privé. Le constructeur Singleton a une visibilité privée ce qui empêche l'instanciation de sa classe ailleurs.

Un point d'accès global. L'accès à une instance de la classe Singleton se fait à l'aide de la méthode getInstance(). L'accès à cette méthode se fait d'une manière statique. Ceci relève du fait qu'on ne peut

pas instancier la classe Singleton et pour appeler cette méthode on y accède directement à partir de sa classe et non pas à partir d'une instance de sa classe.

La méthode getInstance() ne permet de récupérer qu'une seule instance de la classe Singleton.

Singleton à chargement tardif

On appelle Singleton à chargement tardif lorsque l'initialisation de la variable de classe singleton ne se fait pas au moment de chargement de programme, mais plutôt, au moment du premier appel de la méthode getInstance() (voir programme 19). La méthode d'accès globale dans cette version est de la forme suivante :

```
if (singleton == null) { singleton = new Singleton();}
```

```
class Singleton {

  private static Singleton uniqueInstance;

  private Singleton() {       }

  public static Singleton getInstance() {

    if (uniqueInstance == null) {
     uniqueInstance = new Singleton();}

    return uniqueInstance;
  }
}
```

Programme 19: Programme Java contenant une instance du patron Singleton à chargement tardif.

Singleton à chargement non tardif

Un patron Singleton à chargement non tardif est un Singleton dans lequel l'initialisation de la variable de classe contenant l'instance du Singleton se fait lors de chargement de classe. Cette version est la version la plus primitive du Singleton. Pour avoir une idée sur cette version voir le programme 20.

```
class Singleton {

  private static Singleton uniqueInstance = new Singleton();

  private Singleton() {       }

  public static Singleton getInstance() {
   return uniqueInstance;
  }
}
```

Programme 20: Programme Java contenant une instance du patron Singleton à chargement non tardif.

2 La transformation du Singleton dans la littérature

L'introduction du patron Singleton dans un programme et sa suppression (*Inline Singleton*) ont été déjà traité par Kerievsky [Ker04]. Nous présentons ici les démarches qu'il propose pour faire ces transformations ainsi que les instances du Singleton qu'il traite.

Introduire le patron Singleton (État de l'art)

Kerievsky [Ker04] propose une démarche qui permet d'introduire une instance du patron Singleton à chargement non tardif dans un programme. Cette démarche est constituée de trois étapes principales comme indiqué dans l'algorithme 14. Nous utilisons la même démarche que lui avec la différence que nous automatisons, formalisons et vérifions cette démarche par une précondition. Nous détaillons cette démarche plus tard dans la section 3.

Programme initial : programme avec Singleton

Nous montrons ci dessous un exemple sur lequel nous illustrons l'application de la transformation proposée par Kerievesky pour introduire un patron Singleton.

```
class Singleton {
 String name;
 public Singleton(String singletonname){
  this.setName(singletonname);
 }
 void setName(String newname){
  name = newname;
 }
 String getName(){
  return  name;
 }
}
```

```
class SingletonUser {
  public void useSingleton(){
   System.out.println(
         new Singleton("mySingleton")+"---"+
               new Singleton("mySingleton").getName());
  }
}
```

Ce programme contient une classe Singleton et qui peut être instanciée sans limite. L'utilisation de cette classe se fait par l'intermédiaire de la méthode getSingleton(String singletonname).

Programme cible : programme avec Singleton

Voici ci-dessous le programme ciblé par l'algorithme proposé par Kerievesky dans lequel la classe Singleton du programme initial est devenu un Singleton.

```
public class Singleton {
 static String name;
 private static Singleton singleton = new Singleton(name);;

 private Singleton(String singletonname){
  this.setName(singletonname);
 }
 public static Singleton getSingleton(String singletonname) {
  singleton.setName(singletonname);
  return singleton;
 }
 static void setName(String newname){
  name = newname;
 }
 static String getName(){
  return  name;
 }
}
```

```
class SingletonUser {
 public void useSingleton(){
  System.out.println(Singleton.getSingleton("mySingleton") +"---"+
                          Singleton.getSingleton("mySingleton").getName());

 }
}
```

Algorithme d'introduction du Singleton [Ker04]

Nous donnons ici la démarche de l'algorithme 14 proposé par Kerievsky pour introduire le patron Singleton dans le programme initial ainsi que ses états intermédiaires.

1. Identifier une classe qui s'instancie plus qu'une seule fois. Puis appliquer l'opération de refactoring *Replace Constructor with method factory* [Fow99] sur la classe qui va devenir un Singleton et créer une méthode qui est chargée de créer une instance de la classe Singleton.
2. Déclarer un champs privé et statique qui va contenir l'instance de la classe qui va devenir Singleton.
3. Rendre la méthode créée dans l'étape 1 affecter au champs déclaré à l'étape 2 une instance de la classe futur Singleton.

Algorithme 14: Algorithme d'introduction d'un Singleton (Kerievsky [Ker04]).

État du programme après l'étape 1.K.

Remarque. Nous utilisons la lettre K pour distinguer les étapes des algorithmes de Kerievsky de nos algorithmes.

```
class Singleton {
 ...
 private Singleton(String singletonname){
  this.setName(singletonname);
 }
 public static Singleton getSingleton(String singletonname) {
```

```
  return new Singleton(singletonname);
 }
 ...
}

class SingletonUser {
 public void useSingleton(){
   System.out.println(Singleton.getSingleton("mySingleton") +"---"+
                              Singleton.getSingleton("mySingleton").getName());
 }
}
```

État du programme après l'étape 2.K.

```
public class Singleton {
 ...
 private static Singleton singleton;
 ...
}
```

État du programme après l'étape 3.K. Dans cette étape, on met le mot clé *static* pour les éléments de la classe Singleton.

```
public class Singleton {
 static String name;
 private static Singleton singleton = new Singleton(name);;

    private Singleton(String singletonname){
         this.setName(singletonname);
    }

    public static Singleton getSingleton(String singletonname) {
        singleton.setName(singletonname);
        return singleton;
    }

    static void setName(String newname){
        name = newname;
    }
    static String getName(){
        return  name;
    }

    }
```

Après l'application de ces trois étapes, l'instanciation de la classe Singleton est limitée à une seule instance. Noter que Kerievsky a mentionné la possibilité d'obtenir aussi un Singleton à chargement tardif si on fait l'initialisation du champs singleton dans la méthode d'accès global.

Suppression du Singleton (État de l'art)

Kerievsky [Ker04] propose une démarche basée en grande partie sur les opérations de refactoring décrites dans Fowler [Fow99] pour supprimer un Singleton d'un programme Java. Il traite une version d'un Singleton à chargement non tardif (voir le programme 20).

Programme initial

Nous montrons ci dessous une simplification de l'exemple que Kerievesky utilise comme source pour son algorithme de transformation. Voici ci dessous le programme de départ :

```
class Singleton {

 private static Singleton singleton = new Singleton();
 static String name;

 private Singleton(){}

 public static Singleton getSingleton(String singletonname) {
  singleton.setName(singletonname);
  return singleton;
 }
 static void setName(String newname){
   name = newname;
 }
 static String getName(){
   return  name;
 }
}
```

```
class SingletonUser {
  public void useSingleton(){
   System.out.println(Singleton.getSingleton("mySingleton")+"---"+
                           Singleton.getSingleton("mySingleton").getName());
 }
}
```

Ce programme contient une classe Singleton qui s'instancie une seule fois. L'utilisation de cette classe se fait par l'intermédiaire de la méthode getSingleton(String singletonname).

Programme cible : programme sans Singleton

Voici ci-dessous le programme ciblé par l'algorithme de suppression du Singleton proposé par Kerievesky :

```
public class SingletonUser {

  public static Singleton singleton = new Singleton();
  static String name;

  static void setName(String newname){
    name = newname;
  }
  static String getName(){
    return  name;
  }
  public  Singleton getSingleton(String singletonname) {
    setName(singletonname);
    return singleton;
  }
  public void useSingleton(){
   System.out.println(getSingleton("mySingleton")+"---"+ getName());
  }
}
```

Algorithme de suppression du Singleton (*Inline Singleton* [Ker04])

Nous donnons ici la démarche de l'algorithme 15 proposé par Kerievsky pour supprimer le patron Singleton du programme initial ainsi que ses états intermédiaires.

I) Copier les méthodes publiques du Singleton dans les classes qui utilisent le Singleton. Ces méthodes doivent être copiées sous forme des délégateurs pour celles du Singleton. On peut par la suite supprimer le mot clé static dans la classe utilisateur.

II) Changer tous les appels de la méthode du Singleton qui est copiée dans la classe utilisateur vers des appels de la classe utilisateur.

III) Déplacer toutes les autres méthodes et champs du Singleton vers la classe utilisateur et se débarrasser du mot clé static. Cette étape est faite par les deux opérations de refactoring *Move Method* et *Move Field* décrite dans Fowler [Fow99].

IV) La dernière étape consiste à supprimer le Singleton vu qu'on n'en a pas besoin. Mais avant de le supprimer il faut faire un *inline* pour la méthode qui est déléguée dans la classe utilisateur.

Algorithme 15: Algorithme de suppression d'un Singleton (Kerievsky [Ker04]).

Remarque. Noter que l'étape de *inline* de la méthode qui est délégateur de la méthode globale du Singleton n'est pas aussi mentionnée dans l'étape IV. Ceci doit être fait avant la suppression du Singleton, car il y a une méthode de la classe utilisateur du Singleton qui contient encore un appel vers une méthode de la classe Singleton.

État du programme après l'étape I.K.

```
class SingletonUser {
 public  Singleton getSingleton(String singletonname) {
  return Singleton.getSingleton(singletonname);
 }
 public void useSingleton(){
  System.out.println(Singleton.getSingleton("mySingleton")+"---"+
                          Singleton.getSingleton("mySingleton").getName());
 }
}
```

État du programme après l'étape II.K.

```
class SingletonUser {
  public  Singleton getSingleton(String singletonname) {
   return Singleton.getSingleton(singletonname);
  }
  public void useSingleton(){
   System.out.println(getSingleton("mySingleton")+"---"+
                             getSingleton("mySingleton").getName());
  }
}
```

État du programme après l'étape III.K.

```
class SingletonUser {
  public static Singleton singleton = new Singleton();
  String name;
  void setName(String newname){
      name = newname;
   }
   String getName(){
     return  name;
   }
   public  Singleton getSingleton(String singletonname) {
     return Singleton.getSingleton(singletonname);
   }
   public void useSingleton(){
     System.out.println(getSingleton("mySingleton")+"---"+ getName());
   }
}
```

État du programme après l'étape IV.K.

```
class SingletonUser {
 ...
 public  Singleton getSingleton(String singletonname) {
  setName(singletonname);
  return singleton;
 }
...
}
```

3 Démarche suivie

La démarche de Kerievsky [Ker04] adoptent les points suivants :

1. Des algorithmes écrits en langage naturel.
2. Une version basique du patron Singleton.
3. Des transformations non pas encore validées par une précondition.
4. Les deux algorithmes permettent une transformation réversible sur un même programme mais cette caractéristique n'est pas mise en valeur dans cette approche.

Nous traitons ces quatre points comme suit :

1. Nous définissons des opérations de refactoring qui ne sont pas disponibles dans l'outil de refactoring et qui sont nécessaires pour le transformation de l'instance du Singleton que nous traitons.
2. Nous fournissons deux algorithmes formalisés qui s'appliquent sur une version du Singleton à chargement tardif et non pas seulement sur une simple version basique. Ces deux algorithmes assurent une transformation réversible pour un seul programme.
3. Nous générons la précondition qui garantit la réussite des deux algorithmes de transformation.

4 Définitions et formalisation des opérations de refactoring utilisées

Nous présentons ici les opérations qui ne sont pas offertes par l'outil de refactoring et qui sont nécessaires pour la transformation d'une instance de Singleton à chargement tardif. Ces opérations sont utilisées dans les algorithmes 17 et 16 qui servent à supprimer/introduire le patron Singleton et qui seront détaillées plus tard. Les autres opérations qui sont utilisées dans ces algorithmes et qui ne sont pas présentées ici sont détaillées dans l'annexe B.

L'opération *InitializeField*

Rôle. *InitializeStaticField(c,f,t,v)* : cette opération sert à initialiser le champs $c :: f$ qui est de type t par la valeur v. Cette opération est à usage restrictive et elle est conçue pour traiter des cas très spécifiques. Cette restriction est décrite par la précondition de cette opération qui est détaillée ci-dessous. Voici ci-dessous deux exemples qui illustrent l'utilisation de cette opération. L'exemple de la figure 7.1 montre une utilisation correcte de cette opération. Il s'agit ici de faire passer l'instance de la classe C à la variable de classe cField dans une phase plus amant que celle du programme initial sans changer la valeur de cField qui sera initialisée une seule fois. Dans l'exemple de la figure 7.2, l'initialisation du champs cField par une instance de la classe C change la sémantique car la variable cField va changer de valeur plus tard avec une nouvelle instance de cette classe, or l'objectif du programme est d'initialiser la variable de classe une et une seule fois par une seule instance de la classe C. L'exemple suivant illustre l'effet de cette opération :

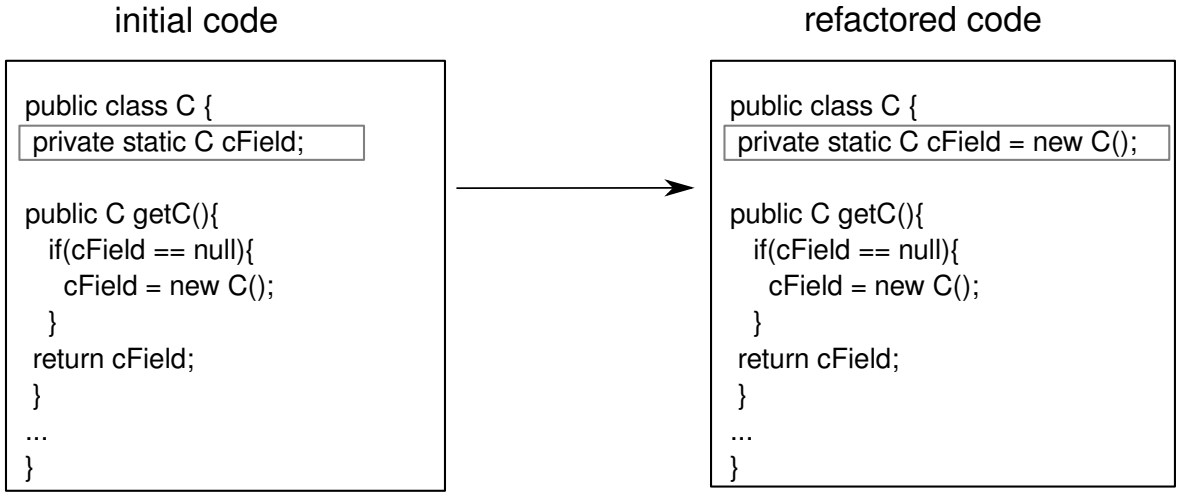


FIGURE 7.1: Exemple d'utilisation correcte de l'opération *InitializeStaticField(c,f,t,v)*

Précondition. Voici la précondition de l'opération *InitializeField* :

$$
\begin{aligned}
&\texttt{ExistsClass}(c)\\
&\wedge\ \texttt{ExistsField}(c, f)\\
&\wedge\ \texttt{IsPrivate}(c, f)\\
&\wedge\ \texttt{HasType}(v, t)\\
&\wedge\ \neg\texttt{IsInitialized}(c, f)\\
&\wedge\ \texttt{IsStatic}(c, f)\\
&\wedge\ (\neg\texttt{IsFieldValueModified}(c, f)\\
&\quad\ \vee\ \texttt{FieldhasValue}(c, f, v))
\end{aligned}
$$

Cette précondition doit vérifier les propriétés suivantes :

- Le champs f existe dans la classe c.

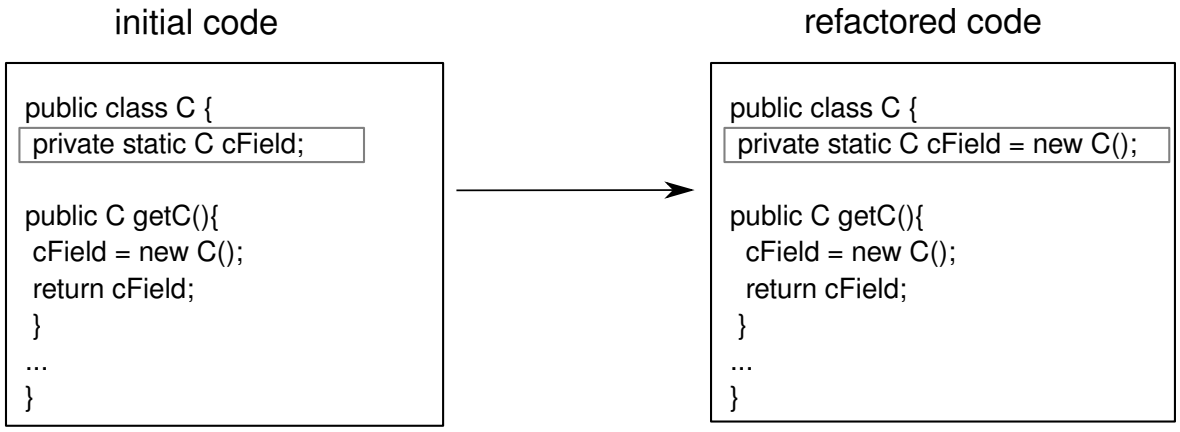


FIGURE 7.2: Exemple d'utilisation fausse de l'opération *InitializeStaticField(c,f,t,v)*

$\texttt{ExistsClass}(c)$
$\wedge\, \texttt{ExistsField}(c, f)$

- Pour restreindre l'application de l'opération *InitializeField*, nous l'appliquons uniquement aux champs qui sont privées et statiques. La propriété privée garantit que le champs en question n'est pas utilisé en dehors de la classe c. La propriété statique est vérifiée car nous appliquons cette opération que sur les variables de classe ou sur une variable utilisée dans un contexte statique. Ces propriétés sont décrites par :
$\texttt{IsPrivate}(c, f)$
$\wedge\, \texttt{IsStatic}(c, f)$

- Le champs $c :: f$ n'est pas déjà initialisé ou initialisé à la valeur nulle. Ceci est vérifié par la précondition :
$\neg\texttt{IsInitialized}(c, f)$

- La valeur à affecter au champs $c :: f$ doit être de même type que celui de ce champs :

$\texttt{HasType}(v, t)$

- Nous avons mentionné que le champs à initialiser est privée mais ceci n'empêche pas de pouvoir modifier sa valeur dans la même classe où il est déclaré. Pour résoudre ce problème, nous vérifions que si le champs f est utilisé en écriture dans la classe c alors elle la valeur qui lui est affecté est égale à v :

$\neg\texttt{IsFieldValueModified}(c, f)$

$\vee\, \texttt{FieldhasValue}(c, f, v))$

Rétro-description.
$\texttt{IsInitialized}(c, f) \mapsto \top$
$\texttt{FieldhasValue}(c, f, v) \mapsto \top$
$\texttt{isConditionSatistfied}(c, M, C, (f == null)) \mapsto \bot$
L'application de l'opération *InitializeField* est décrite par la rétro-description suivante :

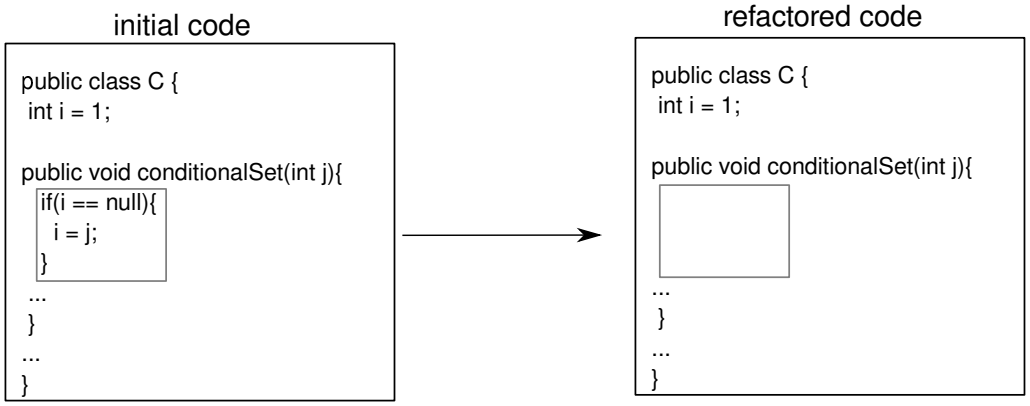
- Le champs $c :: f$ est initialisé par la valeur v :

$\texttt{IsInitialized}(c, f) \mapsto \top$
$\texttt{FieldhasValue}(c, f, v) \mapsto \top$

- Toute condition qui vérifie que le champs $c :: f$ est initialisé à la valeur nulle sera fausse :
  $\texttt{isConditionSatistfied}(c, M, C, (f == null)) \mapsto \bot$
  Nous aurions du définir un prédicat `IsNull` pour vérifier cette propriété. Mais nous voulons expliciter le test sur la condition (f== null) car cette règle de calcul est nécessaire pour d'autres opérations de l'algorithme lorsque nous calculons la précondition de la transformation.

L'opération *DeleteUnusedConditionalStructure*

Rôle. *DeleteUnusedConditionalStructure(c,m,cs,cn)* : cette opération est utilisée pour supprimer la structure conditionnelle cs qui se trouve dans le corps de la méthode $c :: m$ et dont sa condition est cn. La structure conditionnelle à supprimer n'est pas utilisée et n'a aucun effet sur le comportement du programme et donc sa suppression ne viole pas la sémantique. Voici ci dessous une exemple d'application de cette opération :



Préconditions.
$\texttt{ExistsClass}(c)$
$\wedge\, \texttt{ExistsMethodDefinition}(c, m)$
$\wedge\, \texttt{ExistsCode}(c, m, cs)$
$\wedge\, \neg\texttt{isConditionSatistfied}(c, m, cs, cn)$
Pour que cette opération réussisse, nous devons vérifier les propriétés suivantes :

- L'existence du code qui représente la structure conditionnelle dans le corps de la méthode $c :: m$: $\texttt{ExistsCode}(c, m, cs)$
- La condition cn de la structure conditionnelle cs est toujours fausse. Cette propriété garantit que les instructions de la structure conditionnelle ne seront jamais exécutées et par la suite cette structure conditionnelle n'a aucun effet sur le comportement de programme. Cette propriété est vérifiée par :

  $\neg\texttt{isConditionSatistfied}(c, m, cs, cn)$

Rétro-description.
$\texttt{ExistsCode}(c, m, cs) \mapsto \bot$

Ceci signifie que la structure conditionnelle n'existe plus dans le corps de la méthode $c :: m$ après l'application de l'opération *DeleteUnusedConditionalStructure*

L'opération *CreateLazyLoading*

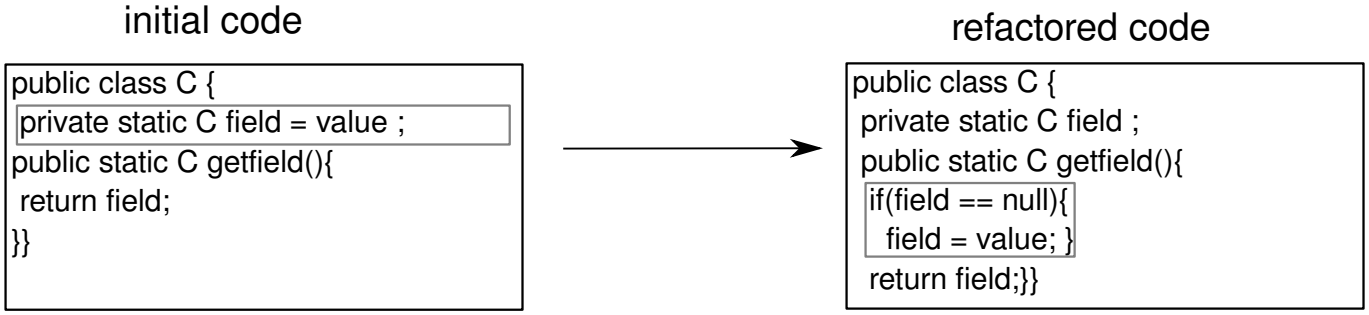
Rôle. *CreateLazyLoading(c,m,f,t,v)* : soit le programme suivant :

```
class C {
 private static C field = value;
 public static C getfield(){
  return field;
 }
}
```

Nous remarquons que l'initialisation du champs f par la valeur value se fait au moment du chargement du programme. Notre objectif est de rendre cette initialisation paresseuse lors du premier appel de la méthode getfield. Pour préserver la sémantique nous devrons garantir que cette initialisation paresseuse se fait une seule fois même si on appelle la méthode getfield plusieurs fois. Pour faire ceci, on met cette initialisation sous une condition de la façon suivante :

```
class C {
 private static C field;
 public static C getfield(){
  if (field == null) field = v;
  return field;
 }
}
```

La condition field == null permet à l'initialisation field = v d'avoir lieu une seule fois à partir du premier appel de la méthode getfield. L'opération *CreateLazyLoading(c,m,f,t,v)* est utilisée donc pour annuler l'initialisation du champs $c :: f$ par la valeur v au moment du chargement du programme et de créer une initialisation paresseuse dans le corps de la méthode $c :: m$ et qui sera déclenchée une seule fois lors du premier appel de cette méthode.



Précondition.
$\texttt{ExistsClass}(c)$
$\wedge\, \texttt{ExistsMethodDefinition}(c, m)$
$\wedge\, \texttt{ExistsField}(c, f)$
$\wedge\, \texttt{IsStatic}(c, f)$
$\wedge\, \texttt{IsInitialized}(c, f)$
$\wedge\, \texttt{IsPrivate}(c, f)$
$\wedge\, \texttt{FieldhasValue}(c, f, v)$ Cette précondition vérifie que le champs $c :: f$ est privée, initialisé par la valeur v et statique. En effet, on vérifie que l'initialisation se fait au moment du chargement de programme.

Rétro-description.
$\texttt{ExistsCode}(c, m, if(f == null)f = v;) \mapsto \top$
$\texttt{FieldhasValue}(c, f, null) \mapsto \top$
$\texttt{IsInitialized}(c, f) \mapsto \bot$

L'application de l'opération *CreateLazyLoading* est décrite par :

- Une création d'une structure conditionnelle à une seule branche qui va encapsuler l'initialisation du champs f par v dans le corps de la méthode $c :: m$. Ceci se manifeste dans la rétro-description suivante :

  $\texttt{ExistsCode}(c, m, if(f == null)f = v;) \mapsto \top$

- Le champs c : :f est initialisé uniquement à une valeur nulle :

  $\texttt{FieldhasValue}(c, f, null) \mapsto \top$
  $\texttt{IsInitialized}(c, f) \mapsto \bot$

  Le champs c : :f ne sera pas initialisé que lors du premier appel de la méthode $c :: m$.

5 Transformation proposée du patron Singleton

Introduction du patron Singleton dans le programme

Nous appliquons l'algorithme 16 sur le programme 21 pour y introduire une instance du Singleton. Nous détaillons ici les étapes de cet algorithme.

I) *ReplaceConstructorWithfactoryMethod*(singleton,singleton,[],singletonGetter)

II) *ExtractField*(singleton,singletonGetter,[],instanceField,singleton,value)

III) *CreateLazyLoading*(singleton,singletonGetter,instanceField,singleton,value)

Algorithme 16: Algorithme pour introduire un Singleton

Étape I. Nous considérons que la classe Singleton est instanciée plusieurs fois et nous voulons limiter son instanciation par une instance du patron Singleton. Nous rappelons l'état initial de la classe Singleton et de la classe qui l'utilise :

```
class Singleton {
  String name;
  public Singleton(String singletonname){
   this.setName(singletonname);
  }
  void setName(String newname){
   name = newname;
  }
  String getName(){
   return  name;
  }
}

class SingletonUser {
 public void useSingleton(){
  System.out.println(new Singleton("mySingleton")+"---"+
                                  new Singleton("mySingleton").getName());
 }
}
```

```
class Singleton {
 String name;
 public Singleton(String singeltonname){
  this.setName(singeltonname);
 }
 void setName(String newname){
  name = newname;
 }
 String getName(){
  return  name;
 }
}
```

```
class SingletonUser {
 public void useSingleton(){
  System.out.println(new Singleton("mySingleton")+"---"+
                     new Singleton("mySingleton").getName());
 }
}
```

```
class Client {
    public static void main(String[] args){

        SingletonUser singletonUser = new SingletonUser();
        singletonUser.useSingleton();

    }
}
```

Programme 21: Programme résultat de l'introduction du Singleton

L'objectif de cette étape est de rendre le constructeur de la classe Singleton privée. L'accès à une instance de la classe Singleton se fait par l'intermédiaire d'une méthode statique qui retourne une instance de cette classe. Toute instanciation de la classe Singleton sera remplacée par un appel statique vers cette méthode. Cette étape est effectuée par l'application de l'opération de refactoring *ReplaceConstructorWithFactoryMethod*. Voici ci-dessous l'état du programme après l'application de cette étape :

```
class Singleton {
 String name;
 private Singleton(String singletonname){
   this.setName(singletonname);
 }
 public static Singleton getSingleton(String singletonname) {
  return new Singleton(singletonname);
 }
 void setName(String newname){
  name = newname;
 }
 String getName(){
  return  name;
 }
}


class SingletonUser {
  public void useSingleton(){
   System.out.println(Singleton.getSingleton("mySingleton") +"---"+
                            Singleton.getSingleton("mySingleton").getName());
  }
}
```

Á ce stade de la transformation, nous n'avons pas aucune instance du Singleton dans le programme : chaque appel pour la méthode getInstance crée une instance de la classe Singleton.

Étape II. Pour activer l'effet du patron du Singleton sur le programme résultant de l'étape précédente, on doit limiter l'instanciation de la classe Singleton à une seule instance. Pour effectuer ceci :

1. on déclare une variable privée et statique dans la classe Singleton et on le renomme singleton.
2. on initialise la variable de classe singleton par new Singleton() dans le corps de la méthode getInstance.
3. on remplace l'instance de la classe Singleton qui existe dans le corps de la méthode getInstance par la variable singleton

Ces trois sous-étapes se font à l'aide de l'opération *ExtractField* qui est disponible dans l'outil de refactoring. L'état de programme après l'application de cette étape est comme suit :

```
class Singleton {
  private static Singleton singleton;
  String name;

  private Singleton(String singletonname){
   this.setName(singletonname);
  }
  public static Singleton getSingleton(String singletonname) {
   singleton = new Singleton(singletonname);
   return singleton;
  }
  void setName(String newname){
   name = newname;
  }
  String getName(){
   return  name;
  }
}
```

Ce programme montre l'existence d'une instance du patron Singleton représentée par la classe Singleton. Le Singleton figurant dans ce programme est une version basique de ce patron. Pour récupérer la version à chargement tardive nous continuons la transformation avec l'étape III.

Étape III. Pour récupérer le programme Gamma et al. [GHJV95] et obtenir un Singleton à chargement tardif, on doit retarder l'initialisation de la variable de classe singleton. L'initialisation sera faite au sein de la méthode getInstance. Mais cette initialisation ne doit se faire qu'une seule fois ce qui impose de protéger cette initialisation par une condition qui permet d'initialiser la variable singleton que lorsque elle a une valeur nulle. Nous utilisons pour faire ceci l'opération *CreateLazyLoading* et nous obtenons un Singleton à chargement tardif :

```
class Singleton {
 private static Singleton singleton;
 String name;

 private Singleton(String singletonname){
  this.setName(singletonname);
 }
 public static Singleton getSingleton(String singletonname) {
  if(singleton == null){
```

```
        singleton = new Singleton(singletonname);
      }
     return singleton;
     }
     void setName(String newname){
      name = newname;
     }
     String getName(){
      return  name;
     }
    }
```

Suppression du patron Singleton

Programme avec Singleton

Le programme 22 est le sujet de notre transformation qui vise à enlever le Singleton. La version du Singleton figurant dans ce programme est une version à chargement tardif : la création d'une instance de la classe Singleton ne se fait pas au démarrage de l'application mais elle se fait plus tard par l'initialisation de l'instance singleton lors du premier appel de la méthode getInstance(). Nous avons déjà donné une idée sur cette version du Singleton dans la section 1. L'implémentation de cette version du Singleton est présentée dans le livre de Gamma et al. [GHJV95].

```
class Singleton {
 private static Singleton singleton;
 String name;

 private Singleton(String singeltonname){
  this.setName(singeltonname);
 }
 public static Singleton getSingleton(String singeltonname){
  if(singleton == null){
     singleton = new Singleton(singeltonname);
  }
 return singleton;
 }
 void setName(String newname){
  name = newname;
 }
 String getName(){
  return  name;
 }
}
```

```
class SingletonUser {
  public void useSingleton(){
   System.out.println(Singleton.getSingleton("mySingleton") +"---"+
                                   Singleton.getSingleton("mySingleton"));
  }
}
}
```

```
class Client {
    public static void main(String[] args){
        SingletonUser singletonUser = new SingletonUser();
        singletonUser.useSingleton();
    }
}
```

Programme 22: Programme avec Singleton

Dans le programme 22, la classe SingletonUser joue le rôle d'utilisateur du Singleton.

Programme sans Singleton

Nous voulons obtenir le programme 21 qui a été le sujet de la transformation introduction du Singleton.

Algorithme de suppression du Singleton

Nous utilisons les notations suivantes dans cet algorithme :

Notation	Définition
$\mathbb{S}$	Ensemble des classes qui utilisent le Singleton
$\mathbb{M}$	Ensemble des méthodes dans lesquelles on appelle la méthode d'accès pour le Singleton

I) *InitializeField*(singleton, instanceField, instanceFieldtype,value))

II) *DeleteUnusedConditionalStructure*(singleton,singletonGetter,conditionalstructure,condition)

III) *InlineField*(singleton,instancefield)

IV) *MakeMethodVisibilityPublic*(singleton,singletonGetter)

V) ForAll c in $\mathbb{US}$ do *MoveStaticMethodM*(singleton,singletonGetter, c)

VI) ForAll m in $\mathbb{UM}$, c in $\mathbb{US}$ do *InlineMethod*(c, singletonGetter,m)

Algorithme 17: Algorithme pour enlever l'effet du Singleton

Nous appliquons l'algorithme 17 sur les programmes qui comportent une instance du Singleton à chargement tardif. Voici ci dessous l'explication de chaque étape de cet algorithme qui est appliquée sur le programme 22 :

Étape I. La version du Singleton à version tardif est caractérisée par une structure conditionnelle qui permet de limiter l'instanciation du Singleton une seule fois comme montré dans le code source suivant :

```
class Singleton {

private static Singleton singleton;
 public static Singleton getSingleton(String singletonname){
    if(singleton == null){
      singleton = new Singleton(singletonname);}
   ...
  }
 ...
 ...
}
```

Bien que cette structure conditionnelle a la caractéristique clé du Singleton à chargement tardif, elle cause des complications pour la suppression du Singleton. Cette difficulté se manifeste dans la spécification d'une opération de refactoring qui casse la condition de cette structure conditionnelle. La suppression de la condition de cette structure conditionnelle peut violer la sémantique du programme.

Pour éviter les risques de changement de sémantique du programme, nous supprimons l'effet de cette structure conditionnelle. Nous faisons ceci par l'initialisation de la variable singleton avant qu'elle soit vérifiée par la condition de cette structure conditionnelle (la valeur de cette variable n'est plus à null ce qui rend la condition inutile). L'initialisation de cette variable se fait au moment de chargement du programme comme montré par le code suivant :

```
class Singleton {
 static String name;
 private static Singleton singleton = new Singleton(name);

 private Singleton(String singletonname){
   this.setName(singletonname);
 }
 public static Singleton getSingleton(String singletonname){
  if(singleton == null){
     singleton = new Singleton(singletonname);
  }
  singleton.setName(singletonname);
  return singleton;
 }
 void setName(String newname){
   name = newname;
 }
 String getName(){
  return  name;
 }
}
```

Cette étape nécessite une intervention au niveau de la méthode getSingleton, pour initialiser la variable name avec l'instruction singleton.setName(singletonname) ; afin de conserver la sémantique car le passage de la valeur de cette variable se fait à partir de la méthode getSingleton et si on n'ajoute pas cette instruction la variable name s'initialise à null.

Après l'initialisation de la variable de classe singleton lors de sa déclaration, la condition singleton == null devient fausse car la valeur de singleton n'es plus nulle. Dans la version du Singleton à chargement tardif, ceci est légal car on est sûr que la variable singleton n'est pas utilisée ailleurs et elle est modifiée une seule fois par y affecter une instance de la classe Singleton.

Étape II. Dans l'étape précédente, nous avons vu que la structure conditionnelle présente dans le programme 22 n'a plus aucun effet car sa condition est fausse. Bien que la présence de cette structure conditionnelle n'influence pas le comportement de programme, elle empêche de faire un *inline* pour la variable singleton par l'outil de refactoring qui va détecter une ambiguïté sur la valeur de cette variable comme montré par ce code source :

```
class Singleton {
 ...
 private static Singleton singleton = new Singleton(name);

 public static Singleton getSingleton(String singletonname){
         if (singleton == null) {
               singleton = new Singleton(singletonname);
         }
         singleton.setName(singletonname);
         return singleton;
 }
 ...
}
```

Pour enlever l'ambiguïté à l'outil de refactoring, nous supprimons la structure conditionnelle non utilisée à l'aide de l'opération *DeleteUnusedConditionalStructure* obtenant ainsi le code suivant :

```
class Singleton {
 ...
 private static Singleton singleton = new Singleton(name);

 public static Singleton getSingleton(String singletonname){

         singleton.setName(singletonname);
         return singleton;
 }
 ...
}
```

Á ce stade nous obtenons une version basique du patron Singleton qui est similaire à celle utilisée dans la transformation de Kerievsky [Ker04].

Étape III. Dans le but de remplacer plus tard les appels de la méthode getInstance par des instanciations de la classe Singleton, nous faisons un inline de la variable de classe singleton et nous supprimons cette variable :

```
class Singleton {
 ...
 public static Singleton getSingleton(String singletonname){
     new Singleton(name).setName(singletonname);
     return new Singleton(name);
 }
 ...
}
```

Étape IV. Dans l'étape suivante, nous allons déplacer la méthode getInstance vers les classes qui l'utilisent d'une façon statique. Avant d'effectuer cette tâche nous devons changer la visibilité du constructeur Singleton en publique vu qu'il est utilisé dans le corps de cette méthode. Nous appliquons ainsi l'opération *MakeMethodVisibilityPublic*.

Étape V. Dans le but de se débarrasser de toute utilisation statique de la méthode getInstance, nous déplaçons cette méthode vers les classes qui l'utilisent (dans notre exemple, c'est la classe SingletonUser). Après le déplacement de la méthode getInstance vers la classe SingletonUser, les appels vers cette méthode ne se font plus à partir de la classe Singleton :

```
class SingletonUser {
 static String name;
 public static Singleton getSingleton(String singletonname){
   new Singleton(name).setName(singletonname);
   return new Singleton(name);
 }
 public void useSingleton(){
   System.out.println(getSingleton("mySingleton")+"---"+
               getSingleton("mySingleton").getName());
}
}
```

Étape VI. Après l'application de l'étape précédente, nous pouvons faire un *inline* de la méthode getInstance et de la supprimer comme montré ci-dessous :

```
class SingletonUser {
  String name;
  public void useSingleton(){
    new Singleton(name).setName("mySingleton");
    new Singleton(name).setName("mySingleton");
    System.out.println(new Singleton(name) +"---"+ new Singleton(name).getName()
  }
}
```

6 Précondition minimale de la transformation introduction/suppression du Singleton

Chaines d'opérations de refactoring

Les algorithmes 5 et 6 sont les chaines des opérations de refactoring respectives pour l'instanciation des algorithmes de transformation pour l'introduction du Singleton et sa suppression du programme 22.

```
ReplaceConstructorWithMethodFactory(Singleton,Singleton,[],getInstance) ;
ExtractField(Singleton,getInstance,[],singleton,Singleton,new Singleton()) ;
CreateLazyLoading(Singleton,getInstance,singleton,Singleton,new Singleton())
```

Algorithme 18: Chaine d'opérations de refactoring de l'introduction d'une instance du patron Singleton pour le programme 21

```
InitializeStaticField(Singleton,singleton,Singleton,new Singleton()) ;
DeleteUnusedConditionalStructure(Singleton,getInstance,if (singleton== null)singleton = new Single-
ton() ;,(singleton== null)) ;
InlineField(Singleton,singleton) ;
MakeMethodVisibilityPublic(Singleton,Singleton) ;
MoveStaticMethod(Singleton,[],getInstance,[],SingletonUser) ;
InlineAndDelete (SingletonUser,getInstance,[],useSingleton,[],[])
```

Algorithme 19: Chaine d'opérations de refactoring de la suppression d'une instance du patron Singleton pour le programme 22

Précondition générée pour une transformation introduction/suppression d'un Singleton

Voici ci-dessous la précondition minimale pour une transformation introduction/suppression d'un Singleton. Cette précondition représente le point fixe qui garantit le non-échec et la réversibilité de la transformation indépendamment du nombre d'introduction/suppression du Singleton fait.

$\neg\texttt{IsOverriding}(SingletonUser, getInstance)$
$\wedge\ \neg\texttt{IsOverridden}(SingletonUser, getInstance)$
$\wedge\ \neg\texttt{IsRecursiveMethod}(SingletonUser, getInstance)$
$\wedge\ \texttt{ExistsClass}(SingletonUser)$
$\wedge\ \neg\texttt{ExistsMethodDefinitionWithParams}(SingletonUser, getInstance, [\,])$
$\wedge\ \neg\texttt{IsPrivate}(Singleton, getInstance)$
$\wedge\ \texttt{ExistsMethodDefinition}(Singleton, Singleton)$
$\wedge\ \neg\texttt{IsFieldValueModified}(Singleton, uniqueInstance)$
$\wedge\ \texttt{HasType}(newSingleton(), Singleton)$

$\wedge$ ($\neg$`IsFieldValueModified`($Singleton, uniqueInstance$)
$\vee$ `FieldhasValue`($Singleton, uniqueInstance, newSingleton()$))
$\wedge$ `FieldhasValue`($Singleton, uniqueInstance, newSingleton()$)
$\wedge$ $\neg$`ExistsField`($Singleton, uniqueInstance$)
$\wedge$ `IsValidStatementSelection`($Singleton, getInstance, [], newSingleton()$)
$\wedge$ `ExistsClass`($Singleton$)
$\wedge$ `ExistsMethodDefinitionWithParams`($Singleton, Singleton, []$)
$\wedge$ $\neg$`ExistsMethodDefinition`($Singleton, getInstance$)

Nous interprétons cette précondition comme suit :

Présence du Singleton. Les propriétés suivantes vérifient l'existence d'une instance du patron Singleton :

Existence d'un classe avec un Constructeur privé (la classe Singleton :
`ExistsClass`($Singleton$)
$\wedge$ `IsPrivate`($Singleton, Singleton$)

Existence d'une variable de classe privée et non initialisée (la variable singleton) :
`ExistsField`($Singleton, singleton$)
$\wedge$ `IsPrivate`($Singleton, singleton$)
$\wedge$ `IsStatic`($Singleton, singleton$)
$\wedge$ $\neg$`IsInitialized`($Singleton, singleton$)

Existence d'une méthode d'accès à une seule instance du Singleton (la méthode getInstance) :
`ExistsMethodDefinition`($Singleton, getInstance$)
$\wedge$ `IsStatic`($Singleton, getInstance$)
$\wedge$ `ExistsCode`($Singleton, getInstance, if(singleton == null)singleton = newSingleton();$)

Ces trois propriétés caractérisent une version de patron Singleton à chargement tardif, ce que est le cas pour le programme de départ (programme 22).

Propriétés vérifiées pour la suppression du Singleton. Pour supprimer le Singleton, les opérations de refactoring utilisées doivent vérifier quelques propriétés pour pouvoir s'exécuter :

L'opération *InitializeField*, appliquée dans l'étape I de l'algorithme 17, doit vérifier que le champs singleton n'est pas initialisé au moment de chargement du programme. Elle doit vérifier aussi que si ce champs peut être modifiée lors de l'exécution du programme alors la valeur qui lui sera affectée doit être égale à celle qu'on voudrait l'initialiser avec (dans ce cas la valeur est new Singleton()). Ceci est vérifié par :
$\neg$`IsInitialized`($Singleton, singleton$)
($\neg$`IsFieldValueModified`($Singleton, singleton$)

$\vee$ `FieldhasValue`($Singleton, singleton, newSingleton()$))

Dans l'étape V de l'algorithme 17, l'opération *MoveStaticMethod* qui déplace la méthode getInstance vers la classe SingletonUser doit vérifier que cette classe existe et que la méthode getInstance n'y existe pas :
`ExistsClass`($SingletonUser$)
$\neg$`ExistsMethodDefinitionWithParams`($SingletonUser, getInstance, []$)

L'opération *InlineMethod* est utilisée pour faire un inline d'une méthode puis la supprimer et elle doit vérifier les propriétés suivantes sur la méthode getInstance :

$\neg$`IsOverriding`($SingletonUser, getInstance$)

$\wedge \neg$`IsOverridden`$(SingletonUser, getInstance)$
$\wedge \neg$`IsRecursiveMethod`$(SingletonUser, getInstance)$

Propriétés vérifiées pour l'introduction du Singleton. Dans l'étape II de l'algorithme 16, l'opération *ExtractField* doit vérifier que la partie à extraire pour l'encapsuler dans un champs est valide lorsque on le sélectionne ce qui est le cas si on sélectionne par exemple new Singleton(). Ceci est vérifié par la propriété suivante :

`IsValidStatementSelection`$(Singleton, getInstance, [\,], newSingleton())$

7 Bilan

Nous avons défini deux algorithmes qui permettent de d'introduire et de supprimer un patron Singleton à chargement tardif. La variation que nous traitons est plus difficile à transformer par rapport à celle utilisée dans Kerievsky [Ker04] vu la présence d'une condition sur l'initialisation du Singleton dont l'inhibition ou l'activation sont délicates au niveau refactoring. Nous avons aussi validé notre transformation sur ce patron par une précondition.

8

Conclusion

Sommaire

1 Résultats

Transformation entre patrons à propriétés complémentaires pour les maintenances modulaires

Nous avons défini deux algorithmes qui spécifient les étapes nécessaires pour passer d'un programme structuré selon le patron Composite vers un programme structuré selon le patron Visiteur et qui est lui équivalent de point de vue sémantique. Ce choix de transformation est signifiant pour la génie logiciel où l'optimisation du coût de maintenance est parmi les choses les plus prioritaires. Ceci est pris en compte dans la transformation entre le patron Composite et Visiteur qui ont des propriétés duales de modularité. Le passage entre ces deux patrons permet de bénéficier de deux propriétés différentes de modularité sur un même programme et par la suite bénéficier de deux types de maintenances modulaires.

Chaque algorithme de la transformation est exprimé par une tâche de refactoring où la préservation de la sémantique est vitale. Chaque étape de ces algorithmes est implémentée par l'invocation d'une ou plusieurs opérations de refactoring (tout dépend si l'opération définie est élémentaire ou composée). Ceci se fait à partir de l'API de l'outil de refactoring de IntelliJ IDEA. L'enchainement de ces opérations forme une transformation automatique qui s'applique sur les programmes Java structurés selon une version basique du Composite pour obtenir une structure Visiteur. Le retour est assuré par un deuxième algorithme qui permet de récupérer le programme initial.

Adaptation de la transformation aux variations des patrons Composite et Visiteur

Les patrons de conception peuvent avoir plusieurs implémentations ou plusieurs versions. Par exemple, il y a un visiteur interne dans lequel l'appel récursif vers la méthode *accept* existe dans la partie visitée et il y a un Visiteur externe dans lequel cet appel existe dans la partie visiteur. On peut avoir aussi un Visiteur dans lequel la méthode *accept* est invoquée directement par un Client ou bien invoquée par l'intermédiaire d'un

délégateur. Mais ces variations ne changent pas le rôle du Visiteur à l'exception de quelques différences comme l'effet sur l'optimisation.

Nous avons découvert que les transformations de base que nous avons défini ne s'appliquent pas sur quelques variations des patrons Composite et Visiteur. Cette découverte a eu lieu lorsque nous avons essayé de valider la transformation de base sur le programme JHotDraw. Lors de ce test, nous avons pu identifié quatre variations dans une instance du patron Composite existante dans ce programme. Cette instance du Composite comporte des méthodes avec paramètres, des méthodes ayant des types de retour différents, une hiérarchie à plusieurs niveaux et une interface au lieu d'une classe abstraite.

Nous avons étudié chaque variation indépendamment de l'autre et nous avons pris les mesures nécessaires pour adapter les algorithmes de base pour être appliqués à ces variations. Le travail important qui a été fait dans ce cadre était la réutilisation des algorithmes de base. La difficulté résidait dans les points suivants :

Localisation. Il s'agit de localiser l'endroit où nous devrons intervenir dans l'algorithme de base pour traiter la variation du patron en question. Ce point est critique pour certains variations : par exemple dans la variation hiérarchie avec plusieurs niveaux, lorsque nous appliquons la transformation aller, nous devons appliquer en premier lieu l'étape 1.C pour avoir une hiérarchie dépourvue de définitions aléatoires des méthodes métier, puis nous appliquons l'algorithme de base tel qu'il est. Ceci permet de minimiser l'interférence avec les autres variations si le programme contient plus qu'une variation.

Identification. Il s'agit d'identifier la bonne opération qui va remplacer l'étape qui cause problème dans l'algorithme de base ou l'ajouter à l'algorithme de base. Dans ce contexte, nous avons identifié cinq nouvelles opérations de refactoring.

Greffe et mise à jour. Il s'agit de greffer la nouvelle opération dans l'algorithme de base et étudier si cette greffe influence les autres étapes de l'algorithme. Par exemple, dans la variation méthodes avec paramètres, l'étape 4.A remplace deux étapes de l'algorithme de base (les étapes 1 et 4).

Validation de la transformation Composite $\leftrightarrow$ Visiteur

Validation à petite échelle. L'automatisation de la transformation basée sur le refactoring est importante pour accélérer le passage d'une structure à l'autre et pour préserver la sémantique. Mais le passage de cette transformation à grande échelle devient critique si on ne garantit pas le bon déroulement de la transformation. Pour éviter ce risque, nous avons spécifié chaque opération de refactoring des différents algorithmes de la transformation (version de base plus les variations), puis nous avons utilisé cette spécification dans un système de calcul pour générer la précondition qui garantit le non échec, la réversibilité et la préservation de la sémantique de ces transformations avant leurs lancements. Nous avons spécifié pour cet effet 24 opérations de refactoring avec un total de 480 règles de calcul.

Les préconditions générées pour les transformations des variations montrent bien les traces de l'adaptation des algorithmes de base que nous avons fait par l'apparition ou la disparition de certaines préconditions. Elles valident aussi la réutilisation que nous avons fait sur les algorithmes de base par le non changement de certaines préconditions vérifiées par ces algorithmes.

Validation à grande échelle. Les variations étudiées dans cette thèse étaient identifiées lors d'une expérimentation de la transformation de base sur JHotDraw ce qui a fait un bon exemple pour valider notre transformation Composite $\leftrightarrow$ Visiteur. L'application d'une telle transformation sur cette application connue, met en valeur son utilisation dans la réalisation des maintenances modulaires. Pour montrer l'intérêt de notre transformation sur JHotDraw, nous avons appliqué un petit scénario de maintenance qui consiste à effectuer une évolution sur une méthode métier de JHotDraw. Pour effectuer cette évolution d'une façon modulaire, nous avons appliqué la transformation Composite $\rightarrow$ Visiteur, nous avons effectué l'évolution voulue sur un seul module et enfin, nous avons appliqué la transformation Visiteur $\rightarrow$ Composite. Le retour vers la structure initial était accompagné par la génération du code ajouté dans la structure Visiteur dans les bons endroits de la structure initiale.

Cette transformation a été aussi validée par la génération d'une précondition qui comporte 1844 prépositions et une chaine de transformation aller-retour comportant 944 occurrences d'opérations de refactoring.

Transformation du patron Singleton

Nous avons défini deux algorithmes de transformation qui permettent d'introduire et supprimer le patron Singleton sur un même programme. Cette transformation nous a guidé à définir des opérations de refactoring qui ne sont pas fournies par les outils de refactoring. Ces opérations sont spécifiés par les préconditions nécessaires pour valider leurs utilisations. Nous avons généré aussi la précondition de point fixe pour la transformation introduction/suppression du Singleton.

Parmi les leçons tirées de cette transformation est que la dualité n'existe pas uniquement au niveau structurale et architecturale mais aussi au niveau comportemental. Tandis qu'un programme avec Singleton met des restrictions sur l'instanciation de la classe Singleton avec des bénéfices en optimisation de mémoire, la suppression du Singleton de ce programme donne plus de souplesse pour la création des nouveaux objets mais avec des coût supplémentaires de mémoires. Nous parlons ici d'une dualité optimisation/souplesse dont la première est intéressante lorsque nous utilisons une seule instance d'une classe donnée et dont la deuxième est intéressante si on veut être plus ouvert suite à des nouvelles exigences du logiciel. Et par la suite, une transformation réversible sur un même programme avec ou sans Singleton permet de bénéficier des deux propriétés sans intervenir à la main pour inhiber ou activer la propriété Singleton.

Intérêts pédagogiques

L'étude de la transformation des patrons met l'accent sur des leçons que nous pouvons tirer au profit de l'enseignement des patrons de conception. Ces transformations permettent de comprendre mieux les patrons de conceptions qui sont difficile à comprendre par une grande partie des élèves de génie logiciel. Par exemple, les variations des patrons Composite et Visiteur peuvent aider les gens à comprendre la complémentarité entre les patrons et que chaque variation dans un patron lui correspond une structure bien précise de l'autre patron. Le patron Composite qui comporte des méthodes avec des types de retour différents par exemple, est duale avec un Visiteur avec types génériques. Parmi les bons exercices qui peuvent être proposés aux étudiants du génie logiciel, on peut leur donner des variations du patron Composite et leur demander d'implémenter la bonne structure du Visiteur qui lui correspond. Ceci permet de prendre conscience que les variations d'un patron de conception peut influencer la structure complémentaire à ce patron.

2 Limites

Certaines limites restent encore à traiter et prendre en considération :

Non-Automatisation complète de certaines transformations. Certaines opérations de refactoring ne sont pas disponibles dans les outils de refactoring ce qui rend quelques transformations semi-automatique. A ce stade du travail, nous n'avons pas implémenté ces opérations dont l'implémentation est délicate et le moindre erreur peut changer la sémantique du programme ce qui demande un temps relativement vaste pour s'occuper de cette tâche.

Vérification manuelle des préconditions. La vérification de la précondition générée sur le programme sur lequel la transformation est appliqué est fait manuellement. Une telle tâche peut être pénible et risquée sur les grands programmes.

La non implémentation des prédicats. Pour alimenter le système de calcul que nous utilisons pour générer la précondition de chaque transformation, nous avons défini un ensemble de prédicats. Ces préconditions nécessitent encore une implémentation pour que nos transformations seront dotées de leurs propres analyses statiques au lieu de l'utilisation des analyses faites par les outils de refactoring.

3 Perspectives

Vers la recherche d'autres patrons à propriétés complémentaires. Nous avons étudié dans cette thèse deux patrons à propriétés complémentaires en terme de modularité et un patron dont sa suppression ou son introduction évoque une dualité en terme d'optimisation. Nous comptons continuer la recherche d'autres patrons à propriétés complémentaires pour mettre la réversibilité des transformations en valeur et bénéficier de différentes structures sur un seul programme. Nous pourrons aussi déduire la complémentarité si l'un des deux patrons peut être remplacé par un autre,comme le cas du Composite que nous pouvons remplacer par le Monteur/*Builder*, et par la suite nous pourrons faire une correspondance entre ce patron et le patron Visiteur. Pour creuser cette piste, nous se basons aussi sur des prémices données dans Gamma et al. [GHJV95] qui portent sur les relations entre certains patrons. Dans ce contexte, nous comptons étudier la nature de ces relations et si elles expriment la complémentarité ou bien elles sont juste des points commun qui relie chaque patron à l'autre.

Inférence de la transformation. La détection des variations d'un patron reste un travail incrémental vu que les variations d'un même patron peuvent être difficile à détecter automatiquement. Ceci influence la transformation d'un patron en y imposant des changements continue ce qui rend la transformation instable. L'inférence de la transformation pour les deux sens reste infaisable pour l'instant. Un premier axe qui pourrait être creusé dans ce contexte est l'inférence d'un seul sens de la transformation à partir de l'autre. Par exemple, on peut définir une transformation d'une variation d'un patron Composite vers un Visiteur, puis on génère automatiquement le sens retour de la transformation de cette variation.

Modularité de la transformation. Une autre approche qui pourrait être adoptée dans le futur, est la modularisation de la transformation elle même : chercher les parties qui restent stables et les encapsuler dans des modules et agir que sur les parties non stables. Cette approche peut amener à trouver une partie stable pour toutes ou plusieurs variations à la fois.

Implémentation et formalisation des opérations de refactoring. L'implémentation des opérations de refactoring reste primordiale pour faire passer les transformations au monde industriel. Mais, l'implémentation doit être accompagnée par des preuves pour qu'elles soient correctes, ce qui fait partie aussi de nos travaux futur.

Validation sur des évolutions des grands logiciels. Les grands logiciels peuvent subir des évolutions importantes. Certaines évolutions peuvent être faites au cours du temps étape par étape sans se rendre compte que leur totalité n'est qu'une tâche de maintenance qui aurait du faite d'une façon modulaire et en une seule fois. La piste que nous pouvons suivre dans ce contexte est de regarder si ces évolutions peuvent être réalisées à l'aide de notre transformation ou non, ce qui validerait notre approche à plus grande échelle.

Bibliographie

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, OOPSLA/ECOOP '90, pages 303–311, New York, NY, USA, 1990. ACM. 11

[BCH+06] David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Trans. Softw. Eng.*, 32(9) :698–717, September 2006. 20

[BJ04] Andrew P. Black and Mark P. Jones. The case for multiple views. In *ICSE 2004 Workshop on Directions in Software Engineering Environments*, 2004. 18

[BKVV08] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Sci. Comput. Program.*, 72 :52–70, June 2008. 166

[BL71] L. A. Belady and M. M. Lehman. Programming system dynamics, or the meta-dynamics of systems in maintenance and growth. Technical report, IBM T.J. Watson Research Center, 1971. 11

[Bru03] K. B. Bruce. Some challenging typing issues in object-oriented languages. *Electronic Notes in Theoretical Computer Science,*, 82, 2003. 13

[BS02] Paulo Borba and Sérgio Soares. Refactoring and code generation tools for AspectJ. In *OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development*, November 2002. 20

[BT95] Don Batory and Lance Tokuda. Automated software evolution via design pattern transformations. Technical report, University of Texas at Austin, 1995. 21

[BT06] Peter Buchlovsky and Hayo Thielecke. A type-theoretic reconstruction of the visitor pattern. *Electronic Notes in Theoretical Computer Science*, 155(0) :309 – 329, 2006. Proc. of the 21st Annual Conf. on Math. Foundations of Prog. Semantics (MFPS XXI). 28

[CC05] G. Canfora and L. Cerulo. How crosscutting concerns evolve in jhotdraw. In *Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on*, pages 65 –73, 0-0 2005. 111

[CD10] Julien Cohen and Rémi Douence. Views, Program Transformations, and the Evolutivity Problem. Research Report hal-00481941, Laboratoire d'Informatique de Nantes Atlantique (LINA) - UMR6241, 2010. 25 pages. http://hal.archives-ouvertes.fr/hal-00481941/en/. 13, 25

[CD11] Julien Cohen and Rémi Douence. Views, Program Transformations, and the Evolutivity Problem in a Functional Language. Research Report hal-00481941, http://hal.archives-ouvertes.fr/hal-00481941/en/, 19 pages, 2011. 20, 158

[CMLC06] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. Multijava : Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst.*, 28 :517–575, May 2006. 13

[CMM+05] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe. A qualitative comparison of three aspect mining techniques. In *IWPC 2005.*, pages 13 – 22, may 2005. 111

[DNS+06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits : A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28 :331–388, March 2006. 11

[Erl00] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2 :17–23, May 2000. 11, 13

[Fow99] Martin Fowler. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley, 1999. 19, 21, 41, 126, 127, 129, 157, 173, 174, 176, 182, 191, 196, 199

[Gar98] J. Garrigue. Programming with polymorphic variants. In *ML Workshop,*, September 1998. 13

[Gar00] J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, November 2000. 13

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995. 26, 28, 33, 35, 37, 123, 137, 138, 148, 157

[GI] Erich Gamma and IFA Informatik. JHotDraw as Open-Source Project. `http://www.jhotdraw.org/`. 15, 111

[GPT12] Rosario Giunta, Giuseppe Pappalardo, and Emiliano Tramontana. Aodp : refactoring code to provide advanced aspect-oriented modularization of design patterns. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1243–1250, New York, NY, USA, 2012. ACM. 20

[HKV12] Mark Hills, Paul Klint, and Jurgen J. Vinju. Scripting a refactoring with rascal and eclipse. In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pages 40–49, New York, NY, USA, 2012. ACM. 19

[HKVDSV11] Mark Hills, Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. A case of visitor versus interpreter pattern. TOOLS'11, pages 228–243. Springer-Verlag, 2011. 25, 166

[HOU03] Stefan Hanenberg, Christian Oberschulte, and Rainer Unland. Refactoring of aspect-oriented software. In *IN PROCEEDINGS OF THE 4TH ANNUAL INTERNATIONAL CONFERENCE ON OBJECT-ORIENTED AND INTERNET-BASED TECHNOLOGIES, CONCEPTS, AND APPLICATIONS FOR A NETWORKED WORLD (NET.OBJECTDAYS*, 2003. 20

[Ker04] Joshua Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004. 25, 26, 124, 125, 126, 127, 129, 130, 141, 144

[KFF98] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 91–113, London, UK, 1998. Springer-Verlag. 13

[KH01] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 313–, New York, NY, USA, 2001. ACM. 11, 19, 20

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag. 20

[KK04] Günter Kniesel and Helge Koch. Static composition of refactorings. *Science of Computer Programming*, 52(Issues 1-3) :9–51, Aug. 2004. 30, 31, 55, 56, 65, 66, 167

[Koc02] Helge Koch. Ein refactoring-framework für Java (in German). Diploma thesis, CS Dept. III, University of Bonn, Germany, April 2002. 173, 176, 182, 196

[KSV09] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal : A domain specific language for source code analysis and manipulation. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 168–177, Washington, DC, USA, 2009. IEEE Computer Society. 25, 166

[Leh96] M. M. Lehman. Laws of software evolution revisited. In *5th European Workshop on Software Process Technology (EWSPT'96)*, volume 1149/1996 of *LNCS*, pages 108–124. Springer, Oct. 1996. 11

[Ler00] Xavier Leroy. A modular module system. *J. Funct. Program.*, 10 :269–303, May 2000. 11

[LH06] Andres Löh and Ralf Hinze. Open data types and open functions. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '06, pages 133–144, New York, NY, USA, 2006. ACM. 13

[LT05] Huiqing Li and Simon Thompson. Formalisation of Haskell Refactorings. In Marko van Eekelen and Kevin Hammond, editors, *Trends in Functional Programming*, Sept. 2005. 20, 74, 160, 165

[LT12] Huiqing Li and Simon Thompson. A domain-specific language for scripting refactorings in erlang. FASE'12, pages 501–515. Springer-Verlag, 2012. 28

[Mar04] Slavisa Marković. Composition of UML Described Refactoring Rules. In *OCL and Model Driven Engineering, UML 2004 Conference Workshop,*, pages 45–59, 2004. 31

[MF05] Miguel P. Monteiro and João M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pages 111–122, New York, NY, USA, 2005. ACM. 20

[MMD07] Marius Marin, Leon Moonen, and Arie van Deursen. An integrated crosscutting concern migration strategy and its application to jhotdraw. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '07, pages 101–110, Washington, DC, USA, 2007. IEEE Computer Society. 20

[MR92] Scott Meyers and Steven P. Reiss. An empirical study of multiple-view software development. *SIGSOFT Softw. Eng. Notes*, 17 :47–57, November 1992. 18

[MT04] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30 :126–139, February 2004. 21, 24, 25, 40

[OC00] Mel Ó Cinnéide. *Automated Application of Design Patterns : A Refactoring Approach*. PhD thesis, Trinity College, Dublin, Oct. 2000. 21, 29, 31, 55

[OWG08] Bruno C.d.S. Oliveira, Meng Wang, and Jeremy Gibbons. The visitor pattern as a reusable, generic, type-safe component. *SIGPLAN Not.*, 43(10) :439–456, 2008. 91

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15 :1053–1058, December 1972. 11

[PJ98] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proceedings of the 22nd International Computer Software and Applications Conference*, COMPSAC '98, pages 9–15, Washington, DC, USA, 1998. IEEE Computer Society. 13

[RBJ97a] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theor. Pract. Object Syst.*, 3 :253–263, October 1997. 19, 155

[RBJ97b] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(Issue 4) :253—-263, 1997. 21, 24, 25

[Saj00] Jorma Sajaniemi. Program comprehension through multiple simultaneous views : A session with VinEd. In *8th International Workshop on Program Comprehension (IWPC)*, pages 99 – 108. IEEE Computer Society, 2000. 18

[SGL07] Macneil Shonle, William G. Griswold, and Sorin Lerner. Beyond refactoring : a framework for modular maintenance of crosscutting design idioms. In *6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 175–184. ACM, 2007. 19

[SI98] J. Sajaniemi and K. Ikonen. Vined - a system for program manipulation through user-definable simultaneous views. *Software - Concepts and Tools*, 19 :130–140, 1998. 18

[Szy02] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. 11

[TCSH06] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design pattern detection using similarity scoring. *IEEE Trans. Softw. Eng.*, 32 :896–909, November 2006. 111

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Jr. Sutton. N degrees of separation : multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 107–119, New York, NY, USA, 1999. ACM. 13, 18

[Tor04] M. Torgersen. The expression problem revisited — four new solutions using generics. In *Proceedings of the 18th European Conference on Object-Oriented Programming,*, June 2004. 13

[VCM+13] Mohsen Vakilian, Nicholas Chen, Roshanak Zilouchian Moghaddam, Stas Negara, and Ralph E. Johnson. A compositional paradigm of automating refactorings. In *ECOOP 2013*, 2013. to appear. 28

[VEdM06] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. Jungl : a scripting language for refactoring. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 172–181, New York, NY, USA, 2006. ACM. 19

[Vis00] Eelco Visser. Language independent traversals for program transformation. Technical report, Universiteit Utrecht, 2000. 19

[Wad87] P. Wadler. Views : a way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 307–313, New York, NY, USA, 1987. ACM. 17

[Wad98] Philip Wadler. The expression problem. Note to Java Genericity mailing list, Nov. 1998. 13

[ZO01] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the International Conference on Functional Programming*, 2001. 13

[ZO05] Mathhias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *Foundations of Object-Oriented Languages (FOOL)*, January 2005. 13

Comparatif des outils de refactoring

Nous présentons ici des outils de refactoring dans le monde des langages objet et fonctionnels.

1 Les outils de refactoring des langages Objet

Nous décrivons ici quelques outils de refactoring qui sont utilisés dans le monde Objet. Nous listons les opérations de refactoring des uns comme SMALLTALK, nous détaillons des opérations de quelques uns (comme les outils de refactoring de ECLIPSE et IntelliJ IDEA[1]) et nous mentionnons juste les noms d'autres outils du monde objet. La concentration sur un outil par rapport à l'autre ici vient du fait qu'elle dépend de l'outil que nous utilisons plus dans cette thèse

Outil de refactoring pour Smalltalk

REFACTORING BROWSER [RBJ97a] est un outil de refactoring utilisé pour effectuer des opérations de refactoring pour le langage SMALTALK. Il est doté d'environ 28 opérations de refactoring. Nous citons parmi ces opérations quelques unes :

- *Rename* : renommer une classe ou une méthode.
- *Move to Component* : déplacer une méthode vers un autre objet.
- *Create subclass* : créer une nouvelle classe dans la hiérarchie existante. Cette opération permet de créer une classe entre un objet et ses sous classes.
- *Extract Method* : extraire un code sélectionné vers une nouvelle méthode.
- *Add Parameter* : ajouter un paramètre à une méthode

Outil de refactoring d'ECLIPSE.

ECLIPSE est un environnement de développement gratuit qui est doté d'un outil de refactoring pour les programmes Java. Le nombre des opérations de refactoring offertes par cet outil atteint les 26 opérations. Nous allons explorer uniquement quelques opérations qui nous intéressent dans cette thèse (voir les tableaux 1, 2 et 3 pour avoir une idée sur la liste de toutes les opérations) :

- *Rename* : renommer une classe, une méthode, une variable, un paquetage ou un dossier.

[1] http ://www.jetbrains.com/idea/

- *Move* : déplacer une méthode d'une classe à une autre.

- *Extract Superclass* : créer une classe mère pour une classe donnée. Les méthodes de la classe en question peuvent être déclarées abstraites dans la classe mère ou bien elles peuvent aussi être montées totalement si c'est possible.

- *Extract Method* : convertir un bloc de code sélectionné en une méthode. Le but de cette opération est de répartir le corps d'une méthode qui peut être long en des autres méthodes dont chacune sera un déléguée pour une partie de ce code. La méthode qui contenait le bloc du code avant d'appliquer cette opération devient un délégateur de la méthode extraite.

- *Change Method Signature* : changer la signature d'une méthode. Son but est de changer l'interface de la méthode en question (par exemple ajout/suppression d'un paramètre). Tous les appels vers cette méthode seront modifiés selon sa nouvelle signature.

Nous présentons ci dessous un exemple d'utilisation de l'opération de refactoring *Extract method* :

Programme initial :

```
class C
 int succ(int x){
   return (x +1);}

 int dispalyAndSucc(int x){
   System.out.println(x);
   return (x +1);          }
}
```

⇒

Après l'application de *Extract method* (avec ECLIPSE) :

```
class C
 int succ(int x){
   return extractedMethod (x);}

 int extractedMethod (int x) {
        return (x +1);     }

 int dispalyAndSucc(int x){
   System.out.println(x);
   return (x +1);          }
}
```

Nous remarquons que la méthode succ est devenu un délégateur pour la nouvelle méthode extraite extractedMethod. Nous remarquons aussi que le code return (x +1) qui se trouve dans le corps de la méthode dispalyAndSucc est égal de point de vue sémantique au code de la méthode extraite et qui peut être remplacé par return extractedMethod (x). En effet, ECLIPSE ne traite pas ces cas lorsque on applique cette opération, au contraire de IntelliJ IDEA que nous allons présenter dans la section suivante.

Outil de refactoring d'IntelliJ IDEA.

IntelliJ IDEA est une plateforme pour développer les programmes Java. Elle comprend une version payante et une autre gratuite. Cette plateforme fournit un outil de refactoring doté d'à peu près 41 opérations de refactoring. Nous avons déjà présenté une idée sur les opérations de refactoring qui nous intéressent dans la section précédente. Ces opérations effectuent les mêmes rôles dans IntelliJ IDEA et ECLIPSE avec quelques différences que nous explorons dans la section 3.

Nous reprenons ici l'exemple de l'opération *Extract Method* pour voir son application avec IntelliJ IDEA. En effet, elle s'applique de la même façon qu'avec ECLIPSE, mais IntelliJ IDEA détecte automatiquement les codes dupliquées de la méthode extraite et il nous propose si on veut remplacer ces codes par un appel vers cette méthode comme montré dans le programme ci-dessous :

Programme initial :

```
class C
 int succ(int x){
   return (x +1);}

 int dispalyAndSucc(int x){
   System.out.println(x);
   return (x +1);          }
}
```

⇒

Après l'application de *Extract method* (avec IntelliJ IDEA) :

```
class C
 int succ(int x){
   return extractedMethod (x);}

 int extractedMethod (int x) {
        return (x +1);     }

 int dispalyAndSucc(int x){
   System.out.println(x);
   return extractedMethod (x);}
}
```

En plus des opérations comparables avec celles de ECLIPSE, IntelliJ IDEA propose aussi des macros opérations qui peuvent introduire des patrons de conception. Par exemple, IntelliJ IDEA propose une opération qui permet d'introduire le patron de conception *Monteur/créateur d'objets* [GHJV95]. Voici ci-dessous un exemple qui montre l'effet de ce refactoring :

Programme initial :

```
class C {
    public C(){}
}
```

⇒

Après l'application de *Replace Constructor With Builder* (avec IntelliJ IDEA) :

```
class C {
    public C(){}
}
class CBuilder {
    public C createC() {
        return new C();
    }
}
```

La richesse de l'outil de refactoring d'IntelliJ IDEA en opérations de refactoring, la réponse rapide aux déclarations des bogues ainsi que que la facilité d'appels des opérations à partir de l'API, nous a motivé à utiliser cet outil dans cette thèse.

2 Autres outils

Il existe d'autres outils de *refactoring* que nous pouvons pas citer tous ici, mais nous citons par exemple quelque uns destinés pour le langage Java : JREFACTORY, JAVAREFACTOR et NetBeans (un IDE pour programmer avec Java et autres langages de programmation).

3 Bilan sur les outils de refactoring des langages à Objet

Les tableaux 1, 2 et 3 montrent un comparatif de quelques outils de refactoring utilisés dans le monde Objet comme celui fourni par IntelliJ IDEA, celui d'ECLIPSE ainsi que d'autres outils de refactoring peu répondus pour Java (juste pour augmenter le degré du comparatif). Notez bien que la colonne **Utilisation dans notre transformation** mentionne si l'opération de refactoring est utilisée dans la transformation que nous avons définie ou non (cette transformation sera détaillée dans le chapitre 3). Le symbole [F] mentionne si l'opération est décrite par Fowler [Fow99] ou non.

Opération	ECLIPSE	NETBEANS	IntelliJ IDEA (version gratuite)	JREFACTORY	JAVA-REFACTOR	**Utilisation dans notre transformation**, chapitres3 et 5
Rename [F]	O	O	O	O	O	O
Change Method Signature [F]	O	O/il fait que la modification des paramètres	O	N	N	O
Pull Up / Pull Member Up [F]	O	O	O	O/Push Up field/method/abstract method	O	O
Push Down / Push Member Down [F]	O	O	O	O/Push Down field/method	O	O
Move method [F]	O	O	O	O	N	O
Move class between paquetages	O	O	O	O	N	N
Extract superclass (une seule classe) [F]	O	O	O	O/Add abstract parent class	N	O
Extract superclass (plusieurs classes) [F]	O	N	N	O	N	O
Extract interface (une seule classe) [F]	O	O	O	O	N	O
Extract interface (plusieurs classes) [F]	N	N	N	O	N	O
Add class	N/ new class	N	N	O	N	N
Use super type where possible	O	O	O/Use Interface Where Possible	N	N	O
In-line [F]	O	N	O	N	N	O

Tableau 1: Tableau comparatif des principaux outils de refactoring des programmes Java (partie 1)

4 Les outils de refactoring des langages fonctionnels

Nous présentons ici deux outils de refactoring utilisés dans le monde des langages fonctionnels. Bien que cette thèse concentre plus sur un langage Objet, mais le commencement de cette thèse est fondé sur une idée de transformation des programmes Haskell dont son outil de refactoring est utilisé par Cohen et Rémi dans leur rapport technique [CD11].

Opération	ECLIPSE	NETBEANS	IntelliJ IDEA (version gratuite)	JREFACTORY	JAVA-REFACTOR	**Utilisation dans notre transformation**, chapitres3 et 5
Replace Method Code Duplicates (Fold)	N	N	O	N	N	O
Extract Method Object	N	N	O	N	N	N
Type Migration	N	N	O	N	N	N
Inline Superclass	N	N	O	N	N	N
Introduce Parameter Object [F]	O	N	O	N	N	O/dans certaines versions
Remove Middleman [F]	N	N	O	N	N	N
Wrap Return Value	N	N	O	N	N	N
Invert Boolean	N	N	O	N	N	N
Safe Delete	O	O	O	O/Remove empty class	N	O
Move Instance Method [F]	N	N	O	N	N	N
Inline Constant [F]	N	N	O	N	N	N
Extract Subclass [F]	N	N	N	N	N	N
Convert To Instance Method	N	N	O	N	N	N
Moving classes and paquetages with reference correction	O	N	O	O/repaquetage pour les classes seulement	N	N
Moving static members with reference correction	O	N	O	N	N	N
Move Inner Class to Upper Level	N	O	O	N	N	N

Tableau 2: Tableau comparatif des principaux outils de refactoring des programmes Java (partie 2)

Nous décrivons ici l'outil de refactoring de Haskell (HaRe) et Wrangler[2] (l'outil de refactoring de Erlang). Ce choix est fait vu que HaRe est un outil qui est doté de deux critères importants qui permettent de réaliser les transformations d'architectures de code source : le premier critère est que HaRe offre des opérations de refactoring variées et nombreuses et le deuxième c'est que ces opérations, en plus qu'elles garantissent la préservation de la sémantique des programmes, elles sont correctes. Nous avons choisi Wran-

[2]les deux outils sont développés à l'université de Kenthttp://www.cs.kent.ac.uk/

Opération	ECLIPSE	NETBEANS	IntelliJ IDEA (version gratuite)	JREFACTORY	JAVA-REFACTOR	**Utilisation dans notre transformation**, chapitres3 et 5
Make Method Static	N	N	O	N	N	N
Copy/Clone Class	N	N	O	N	N	N
Extract Method [F]	O	O	O	O	N	O
Introduce Variable [F] : Introduce Explaining Variable	O	O	O	N	N	N
Introduce Field	O	O	O	N	N	N
Introduce Constant	O	O	O	N	N	N
Introduce Parameter	O	O	O	N	N	N
Extract Class [F]	O	N	O	N	N	N
Replace Inheritance with Delegation[F]	N	N	O	N	N	N
Inline Local Variable [F]	O	N	O	N	N	N
Inline Method [F]	O	N	O	N	N	O
Convert Anonymous Class to Inner	O	O	O	N	N	N
Encapsulate Fields [F]	O	O	O	N	N	O
Replace Temp With Query [F]	N	N	O	N	N	N
Replace Constructor With Factory Method [F]	O	N	O	N	N	N

Tableau 3: Tableau comparatif des principaux outils de refactoring des programmes Java (partie 3)

gler car il comporte aussi plusieurs opérations de refactoring plus ou moins comparables à celles de HaRe.

HaRe.

HaRe [LT05] est un outil de refactoring pour Haskell 98 standard. Il est développé au sein de l'université de Kent. Il a été implémenté avec Haskell. HaRe offre plusieurs opérations de refactoring, mais nous allons nous intéresser dans ce document à présenter celles qui permettent la modification de la structure des programmes Haskell (les opérations de refactoring structurelles). Voici la liste des opérations de refactoring structurelles implémentées dans HaRe :

- *rename* : renommage d'un identifiant.
- *Introduce new def* : introduction d'une nouvelle définition.
- *Generalize def* : généralisation d'une définition.
- *Move def to another module* : déplacer la définition d'une fonction vers un autre module.
- *remove def* : supprime la définition d'une fonction du module courant.
- *Folding definition* : repliage de toute sous expression qui apparait à droite d'une équation contre le nom d'une fonction dont sa définition est la même que ces sous expressions.
- *Unfolding def* : dépliage d'une application de définition.
- *Generative fold* : remplace une expression qui apparait à droite d'une équation par celle qui apparait à gauche de cette éqaution en créant une nouvelle définition récursive.
- *Folding as pattern* : remplace une expression par un *pattern*.
- *Unfolding as pattern* : c'est l'inverse de *Folding as pattern*.
- *Converting from **let** to **where*** : convertit une définition sous la forme de *let* vers une autre sous la forme de *where*.
- *Converting from **where** to **let*** : c'est l'inverse de *Converting from **let** to **where***.
- *Case analysis simplification* : transforme une expression de cas (*case*) en des branches selon le nombre de cas.

Exemples d'opérations de HaRe

Pour montrer l'utilité de HaRe montrons l'utilisation de deux opérations de refactoring :

- *Introduce New Def* : cette opération a été appliquée au programme suivant :

```
f x = x + 1
```

et voici le résultat :

```
f x = g
  where
    g = x + 1
```

- *Lift to top level* : cette opération a été appliquée à la fonction g qui apparait dans le programme résultat de l'exemple précédent. Et voici le résultat de l'application de cette opération :

```
f x = (g x)
g x = x + 1
```

Illustration d'un exemple de composition des opérations de HaRe

La composition des opérations élémentaires de refactoring de HaRe permet d'avoir des différents types de transformations. Comme nous l'avons fait avec la transformation d'un programme structuré selon les données en un autre structuré selon les fonctions, nous avons défini aussi une transformation qui permet d'identifier un aspect de traçage dans un programme Haskell. Le premier type de transformation est intéressant dans les deux sens car le programmeur peut avoir besoin des deux types de maintenances (données/fonctions), tandis que la deuxième est intéressante dans un seul sens (encapsulation de l'aspect du traçage dans un seul module). Bien que le deuxième type de transformation manque d'intérêt par rapport à ce qui était fait dans le monde Objet mais il nous donne comme même un bon départ pour montrer la possibilité d'orchestration des opérations élémentaires de refactoring pour avoir une transformation qui change totalement la structure d'un programme. Ceci vient dans le but d'utiliser ces outils de refactoring non seulement dans des interventions élémentaires (par exemple le renommage) mais aussi de les consacrer pour diminuer le coût de maintenance (dualité données/fonctions) et pour contrôler l'exécution des fonctions à partir d'un seul module (le cas du traçage).

Pour avoir une idée plus claire sur l'effet d'une telle transformation, nous avons un programme initial 23 dans lequel on remarque que le texte destiné à suivre l'exécution des fonctions est mélangé avec le côté fonctionnel du programme. Notre objectif c'est de séparer ce texte du traçage de la partie fonctionnelle du programme. Nous voudrons arriver au programme 24. On remarque que ce programme montre que le traçage n'est plus mélangé avec les fonctions mais plutôt encapsulé dans un module indépendant. Le but de cette transformation c'est d'effectuer des maintenances modulaires sur la partie traçage. Tandis que le premier programme ne les permet pas vu que le programmeur doit toucher des modules différents pour changer le texte, le deuxième les permet.

L'algorithme20 donne une idée sur l'orchestration des opérations de refactoring de HaRe pour passer du programme 23 vers le programme 24

Les opérations utilisées dans l'algorithme 20 se basent chacune sur une opération élémentaire de l'outil de refactoring HaRe.

Wrangler : l'outil de refactoring des programmes Erlang.

Wrangler est un outil de *refactoring* développé aussi au sein de l'université de Kent et destiné pour les programmes Erlang. Wrangler comporte trois types d'opérations de refactoring : structurelles, appliquées sur les processus et appliquées sur les macro. Nous nous intéressons ici à lister les opérations de refactorings structurelles de Wrangler et qui sont :

- *Intro new var* : introduire une nouvelle variable qui va représenter une expression sélectionnée par l'utilisateur.
- *Inline var* : remplacer une variable par sa définition. Cette opération joue le rôle de la fonction inverse si elle est utilisée directement après la fonction *intro new var*.
- *Fun extraction* : introduire une nouvelle fonction qui représente une expression. Cette opération joue le même rôle que l'opération *introduce New Def* de refactorer de haskell si elle est suivie de *lift to toplevel* et de *generalise* dans le refactorer de haskell. Elle fait automatiquement le *lifting* vers le *top level* et la généralisation des variables libres.
- *Rename fun* : renommer une fonction.
- *Rename mod* : renommer un module.
- *Rename var* : renommer une variable.
- *Move fun* : déplacer une fonction d'un module à un autre.
- *Generalise* : généraliser la définition d'une fonction.

```
module Expr  where

data Expr =
     Const Int
   | Add (Expr,Expr)
```

```
module EvalMod where

import Expr
eval :: Expr -> IO (Int)
eval (Const i)     = (do
                       let
                          f =  "evalConst"
                       putStrLn ( "Execution de  "
                                  ++f++" et le resultat est  " ++ show i)
                       return i)

eval (Add(e1 ,e2)) = (do
                       y <- eval e1
                       z <- eval e2
                       let
                         res = y + z
                         f   = "evalAdd"
                       putStrLn ( "Execution de  "
                                   ++f++" et le resultat est  " ++ show res)
                       return res)
```

```
module ToStringMod where

import Expr
toString :: Expr -> IO (String)
toString (Const i)     = (do
                           let f = "toStringConst"
                           putStrLn $ "Execution de  "
                                   ++f++" et le resultat est  " ++ show i
                           return (show i))
toString (Add (e1,e2)) = (do
                           x <- (toString e1)
                           y <- (toString e2)
                           let result = x ++ "+" ++ y
                               f      = "toStringAdd"
                           putStrLn $ "Execution de  "
                                     ++f++" et le resultat est  " ++show(result)
                           return result)
```

Programme 23: Le programme initial

```
module Expr  where

data Expr =
     Const Int
   | Add (Expr,Expr)
```

```
module EvalMod where

import Expr
eval :: Expr -> IO (Int)
eval (Const i)     = (do
                       let
                         f =  "evalConst"
                       logg i f
                       return i)

eval (Add(e1 ,e2)) = (do
                       y <- eval e1
                       z <- eval e2
                       let
                         res = y + z
                         f   = "evalAdd"
                       logg res f
                       return res)
```

```
module ToStringMod where

import Expr
import LogFile (logg)

toString :: Expr -> IO (String)
toString (Const i)     = (do
                            let
                              f = "toStringConst"
                            logg i f
                            return (show i))
toString (Add (e1,e2)) = (do
                            x <- (toString e1)
                            y <- (toString e2)
                            let
                              result = x ++ "+" ++ y
                              f      = "toStringAdd"
                            logg result f
                            return result)
```

```
module LogFile where
 logg i f
     =
         putStrLn
             ("Execution de  " ++
                  (f ++ (" et le resultat est  " ++ (show i))))
```

Programme 24: Le programme cible

- *Tuple funpar* : grouper un ensemble de paramètres d'une fonction dans un tuple .
- *Fold expr* : replier des expressions contre une définition d'une fonction.
- *Unfold fun app* : déplier une application d'une fonction.

Nous présentons un exemple d'utilisation de l'opération de refactoring *Intro new var* (similaire à l'opération *Introduce New Def* de HaRe). Voici le programme Erlang initial :

```
f(X) -> X + 1.
```

Et voici le résultat :

```
(defvar f1          "eval"           )
(defvar f2          "toString"       )
(defvar c1          "Const"          )
(defvar c2          "Add"            )
(defvar f1mod       "EvalMod"        )
(defvar f2mod       "ToStringMod"    )
(defvar loggmod     "LogFile"       )
(defvar logg        "logg"           )

(haskell-refac-newDefIdentApp "putStrLn" logg f1mod)

(haskell-refac-liftOneLevelIn f1 logg f1mod)

(haskell-refac-generalisePatternVariableInLocalDefIn f1 c1 logg f1mod "0" "Curried")
(haskell-refac-liftDefToTopLevelIn f1 logg f1mod)


(haskell-refac-foldToplevelDefinition logg f1mod)

(haskell-refac-moveDefBetweenModules logg f1mod f2mod)
(haskell-refac-foldToplevelDefinition logg f2mod)
(haskell-refac-moveDefBetweenModules logg f2mod loggmod)
(haskell-refac-cleanImportsMod f1mod)
```

Algorithme 20: Algorithme d'extraction du traçage du programme 23.

```
f(X) -> G = X + 1,
        G.
```

Bilan Wrangler vs HaRe (monde fonctionnel)

Le tableau 4 montre une comparaison des opérations de *refactoring* de Wrangler et de HaRe.

Opération de *refactoring* en HaRe	**Son équivalent en Wrangler**	Prouvée
Rename	Rename Function Name/ Rename Variable	Pas d'information
Inroduce New Def	Introduce New Variable	Pas d'information
Generalise Def	Generalise Function Definition	Oui pour HaRe [LT05]/ Pas d'information pour Wrangler
L'enchainement de Inroduce New Def, Generalise et Lift def to toplevel	Function Extraction	Pas d'information
Fold Definition	Fold Expression Against Function	Pas d'information
Unfold Definition	Unfold Function Application	Pas d'information
Move def to another module	Move Definition to Another Module	Oui pour HaRe [LT05]/ Pas d'information pour Wrangler
Duplicate Def into Comment	$\emptyset$	Pas d'information
Remove def	$\emptyset$	Pas d'information
Generative fold	$\emptyset$	Pas d'information

Tableau 4: Tableau comparatif de Wrangler et HaRe

5 Autres Outils de transformation de programmes

Il y a des autres outils qui sont destinés pour la transformation des programmes à part les outils de *refactoring*. Stratego [BKVV08] est un exemple de ces outils : il est basé sur la réécriture des programmes. Stratego offre des règles de réécriture pour permettre aux programmeurs d'écrire leurs transformations. L'outil RASCAL [KSV09] qui est à la base un outil d'analyse des programmes, est utilisé par Hills et al. [HKVDSV11] comme un outil de transformation d'une instance d'un patron de conception Visiteur vers un patron de conception Composite. Nous ne traitons pas ces outils dans cette thèse.

B

Opérations de refactoring

Dans cette annexe, nous spécifions et décrivons les opérations de refactoring que nous utilisons dans nos transformations. Nous identifions pour chacune des opérations les préconditions qu'elle doit vérifier pour qu'elle s'applique.

Les rétro-descriptions (*backward descriptions*) données ici sont utilisées pour alimenter le système de calcul des préconditions de la composition statique des opérations de refactoring proposé par Kniezel et koch [KK04]

1 CreateEmptyClass

Rôle. CreateEmptyClass (classname c) : cette opération est utilisée pour créer une nouvelle classe c.

Outils de refactoring. *new Class* dans Eclipse and IntelliJ IDEA.

Précondition.
$(\neg existsType(c))$

Rétro-description.
$\texttt{ExistsClass}(c) \mapsto \top$
$\texttt{ExistsType}(c) \mapsto \top$
$\texttt{IsAbstractClass}(c) \mapsto \bot$
$\texttt{ExistsMethodDefinition}(c, Y) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(c, Y, [\,]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(c, Y, [T1]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(c, Y, [T1; T2]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(c, Y, [T1; T2; T3]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(c, Y, [T1; T2; T3; T4]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(c, Y, [T1; T2; T3; T4; T5]) \mapsto \bot$
$\texttt{IsInheritedMethod}(c, Y) \mapsto \texttt{IsVisible}(java.lang.Object, Y, c)$
$\texttt{IsInheritedMethodWithParams}(c, Y, [\,])$
$\quad \mapsto \texttt{IsVisibleMethod}(java.lang.Object, Y, [\,], c)$
$\texttt{IsInheritedMethodWithParams}(c, Y, [T1])$

$\mapsto$ IsVisibleMethod$(java.lang.Object, Y, [T1], c)$

IsInheritedMethodWithParams$(c, Y, [T1; T2])$
$\mapsto$ IsVisibleMethod$(java.lang.Object, Y, [T1; T2], c)$

IsInheritedMethodWithParams$(c, Y, [T1; T2; T3])$
$\mapsto$ IsVisibleMethod$(java.lang.Object, Y, [T1; T2; T3], c)$

IsInheritedMethodWithParams$(c, Y, [T1; T2; T3; T4])$
$\mapsto$ IsVisibleMethod$(java.lang.Object, Y, [T1; T2; T3; T4], c)$

IsInheritedMethodWithParams$(c, Y, [T1; T2; T3; T4; T5])$
$\mapsto$ IsVisibleMethod$(java.lang.Object, Y, [T1; T2; T3; T4; T5], c)$

IsSubType$(c, X) \mapsto \bot(condition)$

ExtendsDirectly$(c, X) \mapsto \bot(condition)$

ExistsMethodDefinitionWithParams$(B, Y, [c]) \mapsto \bot$

ExistsMethodDefinitionWithParams$(B, Y, [c; T1]) \mapsto \bot$

ExistsMethodDefinitionWithParams$(B, Y, [T1; c]) \mapsto \bot$

ExistsMethodDefinitionWithParams$(B, Y, [T1; T2; c]) \mapsto \bot$

ExistsMethodDefinitionWithParams$(B, Y, [T1; c; T2]) \mapsto \bot$

ExistsMethodDefinitionWithParams$(B, Y, [c; T1; T2]) \mapsto \bot$

IsInheritedMethodWithParams$(B, Y, [c]) \mapsto \bot$

IsInheritedMethodWithParams$(B, Y, [c; T1]) \mapsto \bot$

IsInheritedMethodWithParams$(B, Y, [T1; c]) \mapsto \bot$

IsInheritedMethodWithParams$(B, Y, [T1; T2; c]) \mapsto \bot$

IsInheritedMethodWithParams$(B, Y, [T1; c; T2]) \mapsto \bot$

IsInheritedMethodWithParams$(B, Y, [c; T1; T2]) \mapsto \bot$

ExistsParameterWithName$(B, Y, [c], P) \mapsto \bot$

ExistsParameterWithName$(B, Y, [c; T1], P) \mapsto \bot$

ExistsParameterWithName$(B, Y, [T1; c], P) \mapsto \bot$

ExistsParameterWithName$(B, Y, [T1; c; T2], P) \mapsto \bot$

ExistsParameterWithName$(B, Y, [T1; T2; c], P) \mapsto \bot$

ExistsParameterWithName$(B, Y, [c; T1; T2], P) \mapsto \bot$

ExistsParameterWithType$(B, Y, [c], P) \mapsto \bot$

ExistsParameterWithType$(B, Y, [c; T1], P) \mapsto \bot$

ExistsParameterWithType$(B, Y, [T1; c], P) \mapsto \bot$

ExistsParameterWithType$(B, Y, [T1; c; T2], P) \mapsto \bot$

ExistsParameterWithType$(B, Y, [T1; T2; c], P) \mapsto \bot$

ExistsParameterWithType$(B, Y, [c; T1; T2], P) \mapsto \bot$

IsUsedMethodIn$(c, B, Y) \mapsto \bot$

IsUsedMethod$(c, B, [T1]) \mapsto \bot$

IsUsedMethod$(c, B, [T1; T2]) \mapsto \bot$

IsUsedMethod$(c, B, [T1; T2; T3]) \mapsto \bot$

IsUsedMethod$(c, B, [T1; T2; T3; T4]) \mapsto \bot$

IsUsedConstructorAsMethodParameter$(c, B, Y) \mapsto \bot$

IsUsedConstructorAsInitializer$(c, B, Y) \mapsto \bot$

IsUsedConstructorAsObjectReceiver$(c, B, Y) \mapsto \bot$

IsUsedConstructorAsMethodParameter$(B, c, Y) \mapsto \bot$

IsUsedConstructorAsInitializer$(B, c, Y) \mapsto \bot$

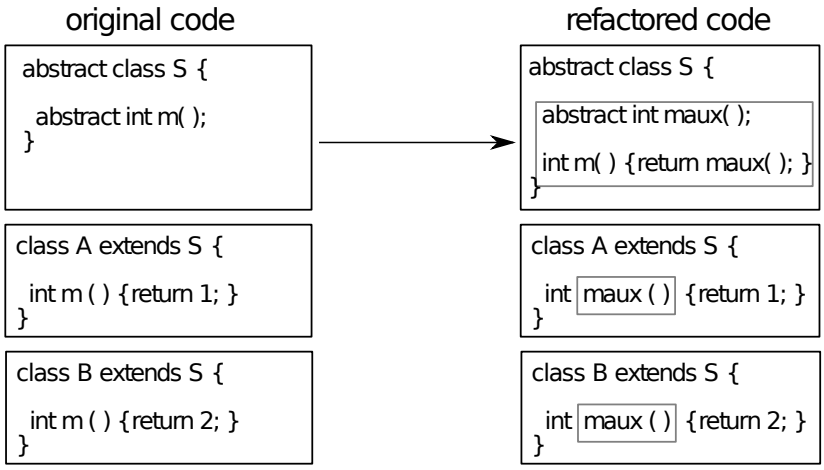
IsUsedConstructorAsObjectReceiver$(B, c, Y) \mapsto \bot$

$\texttt{IsSubType}(B, c) \mapsto \bot$
$\texttt{ExtendsDirectly}(B, c) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(B, Y, [P], [c]) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(B, Y, [P], [c; T1]) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(B, Y, [P], [T1; c]) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(B, Y, [P], [T1; c; T2]) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(B, Y, [P], [T1; T2; c]) \mapsto \bot$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(c, T1, T2, T3) \mapsto \top$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(c, T1, T2) \mapsto \top$
$\texttt{BoundVariableInMethodBody}(c, T1, T2) \mapsto \bot$
$\texttt{ExistsField}(c, F) \mapsto \bot$
$\texttt{ExistsMethodInvocation}(c, Y, T1, X) \mapsto \bot$
$\texttt{ExistsAbstractMethod}(c, Y) \mapsto \bot$
$\texttt{IsIndirectlyRecursive}(c, Y) \mapsto \bot$
$\texttt{IsVisibleMethod}(c, Y, [T1], B) \mapsto \bot$
$\texttt{IsVisibleMethod}(c, Y, [T1; T2], B) \mapsto \bot$
$\texttt{IsVisibleMethod}(c, Y, [T1; T2; T3], B) \mapsto \bot$
$\texttt{IsVisibleMethod}(B, Y, [B], c) \mapsto \bot$
$\texttt{IsVisibleMethod}(B, Y, [B; T1], c) \mapsto \bot$
$\texttt{IsVisibleMethod}(B, Y, [T1; B], c) \mapsto \bot$
$\texttt{IsVisibleMethod}(B, Y, [T1; T2; B], c) \mapsto \bot$
$\texttt{IsVisibleMethod}(B, Y, [T1; B; T2], c) \mapsto \bot$
$\texttt{IsVisibleMethod}(B, Y, [B; T1; T2], c) \mapsto \bot$
$\texttt{IsInverter}(c, Y, T1, T2) \mapsto \bot$
$\texttt{IsDelegator}(c, Y, X) \mapsto \bot$
$\texttt{IsAbstractClass}(c) \mapsto \bot$
$\texttt{IsUsedMethodIn}(c, Y, B) \mapsto \bot$
$\texttt{IsUsedMethodIn}(B, Y, c) \mapsto \bot$
$\texttt{IsPrimitiveType}(c) \mapsto \bot$
$\texttt{IsPublic}(c, Y) \mapsto \bot$
$\texttt{IsProtected}(c, Y) \mapsto \bot$
$\texttt{IsPrivate}(c, Y) \mapsto \bot$
$\texttt{IsUsedAttributeInMethodBody}(c, X, Y) \mapsto \bot$
$\texttt{IsGenericsSubtype}(c, [T1], B, [T2]) \mapsto \bot$
$\texttt{IsGenericsSubtype}(c, [T1; T2], B, [T4; T3]) \mapsto \bot$
$\texttt{IsGenericsSubtype}(c, [T1; T2; T3], B, [T4; T5; T6]) \mapsto \bot$
$\texttt{IsInheritedField}(c, F) \mapsto \bot$
$\texttt{IsOverridden}(c, Y) \mapsto \bot$
$\texttt{IsOverloaded}(c, Y) \mapsto \bot$
$\texttt{IsOverriding}(c, Y) \mapsto \bot$
$\texttt{IsRecursiveMethod}(c, Y) \mapsto \bot$
$\texttt{IsRecursiveMethod}(c, Y) \mapsto \bot$
$\texttt{HasReturnType}(c, Y, T1) \mapsto \bot$
$\texttt{HasParameterType}(c, T1) \mapsto \bot$
$\texttt{HasParameterType}(B, c) \mapsto \bot$
$\texttt{MethodHasParameterType}(c, Y, T1) \mapsto \bot$

$\texttt{AllSubclasses}(c, [C1; C2; C3]) \mapsto \bot$
$\texttt{ExtendsDirectly}(c, java.lang.Object) \mapsto \top$

2 CreateIndirectionInSuperClass

Rôle. CreateIndirectionInSuperclass (s, [a,b], m,[t,t'], r,n) : cette opération est utilisée pour créer un déléguée de la méthode $s :: m$ qui a les paramètres t,t',... et avec le type de retour r et l'appeler n. Les sous classes $a, b, ...$ de s seront influencées par ce changement. Voici ci dessous un exemple de la création d'un déléguée n() d'une méthode int s : :m(). Cette opération est utilisée sur une méthode qui est déclarée dans une classe abstraite et définie sur ses sous classes. Nous supposons aussi que cette opération s'applique sur les méthodes non surchargées.



Outils de refactoring. Avec IntelliJ IDEA :

- Utiliser *Change Signature* sur la méthode m dans la classe s (choisir "delegate via overloading method", spécifier le nouveau nom n, spécifier la visibilité voulue).

Avec Eclipse :

- Utiliser *Change Method Signature* sur la méthode m dans la classe s (choisir "keep original method as delegate to changed method", et spécifier le nouveau nom n).

Précondition.
$(\texttt{ExistsClass}(s)$
$\wedge\ \texttt{IsAbstractClass}(s)$
$\wedge\ \texttt{ExistsMethodDefinitionWithParams}(s, m, [t; t'])$
$\wedge\ \texttt{ExistsAbstractMethod}(s, m)$
$\wedge\ \neg\texttt{IsInheritedMethod}(s, n)$
$\wedge\ \neg\texttt{IsInheritedMethodWithParams}(s, n, [t; t'])$
$\wedge\ \neg\texttt{ExistsMethodDefinitionWithParams}(s, n, [t; t'])$
$\wedge\ \texttt{AllSubclasses}(s, [a; b])$
$\wedge\ \texttt{HasReturnType}(s, m, r)$
$\wedge\ \neg\texttt{IsPrivate}(s, m)$
$\wedge\ \neg\texttt{IsPrivate}(a, m)$
$\wedge\ \neg\texttt{IsPrivate}(b, m)$
$\wedge\ \texttt{ExistsMethodDefinition}(s, m)$
$\wedge\ \texttt{ExistsMethodDefinition}(a, m)$
$\wedge\ \texttt{ExistsMethodDefinition}(b, m)$

$\wedge \neg\texttt{ExistsMethodDefinition}(s, n)$
$\wedge \neg\texttt{ExistsMethodDefinition}(a, n)$
$\wedge \neg\texttt{ExistsMethodDefinition}(b, n))$

Rétro-description.
$\texttt{ExistsAbstractMethod}(s, n) \mapsto \top$
$\texttt{ExistsAbstractMethod}(s, m) \mapsto \bot$
$\texttt{IsDelegator}(s, m, n) \mapsto \top$
$\texttt{IsInheritedMethodWithParams}(s, n, [t; t']) \mapsto \bot$
$\texttt{IsOverriding}(s, n) \mapsto \bot$
$\texttt{ExistsType}(r) \mapsto \top$
$\texttt{HasReturnType}(s, n, r) \mapsto \texttt{HasReturnType}(s, m, r)$
$\texttt{HasReturnType}(a, n, r) \mapsto \texttt{HasReturnType}(s, m, r)$
$\texttt{HasReturnType}(b, n, r) \mapsto \texttt{HasReturnType}(s, m, r)$
$\texttt{ExistsMethodDefinition}(s, n) \mapsto \top$
$\texttt{ExistsMethodDefinition}(a, n) \mapsto \top$
$\texttt{ExistsMethodDefinition}(b, n) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(s, n, [t; t']) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(a, n, [t; t']) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(b, n, [t; t']) \mapsto \top$
$\texttt{ExistsParameterWithName}(s, n, [t; t'], N) \mapsto \bot$
$\texttt{ExistsParameterWithName}(a, n, [t; t'], N) \mapsto \bot$
$\texttt{ExistsParameterWithName}(b, n, [t; t'], N) \mapsto \bot$
$\texttt{ExistsParameterWithName}(s, n, [V], N) \mapsto \texttt{ExistsParameterWithName}(s, m, [V], N)$
$\texttt{ExistsParameterWithName}(a, n, [V], N) \mapsto \texttt{ExistsParameterWithName}(a, m, [V], N)$
$\texttt{ExistsParameterWithName}(b, n, [V], N) \mapsto \texttt{ExistsParameterWithName}(b, m, [V], N)$
$\texttt{ExistsMethodDefinition}(a, m) \mapsto \bot$
$\texttt{ExistsMethodDefinition}(b, m) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(a, m, [t; t']) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(b, m, [t; t']) \mapsto \bot$
$\texttt{IsIndirectlyRecursive}(a, n) \mapsto \texttt{IsRecursiveMethod}(a, m)$
$\texttt{IsIndirectlyRecursive}(b, n) \mapsto \texttt{IsRecursiveMethod}(b, m)$
$\texttt{ExistsMethodInvocation}(a, n, s, m) \mapsto \texttt{IsRecursiveMethod}(a, m)$
$\texttt{ExistsMethodInvocation}(b, n, s, m) \mapsto \texttt{IsRecursiveMethod}(b, m)$
$\texttt{ExistsMethodInvocation}(s, m, a, n) \mapsto \top$
$\texttt{ExistsMethodInvocation}(s, m, b, n) \mapsto \top$
$\texttt{BoundVariableInMethodBody}(s, n, V) \mapsto \texttt{BoundVariableInMethodBody}(s, m, V)$
$\texttt{BoundVariableInMethodBody}(a, n, V) \mapsto \texttt{BoundVariableInMethodBody}(a, m, V)$
$\texttt{BoundVariableInMethodBody}(b, n, V) \mapsto \texttt{BoundVariableInMethodBody}(b, m, V)$
$\texttt{IsOverloaded}(s, n) \mapsto \bot$
$\texttt{IsOverloaded}(a, n) \mapsto \bot$
$\texttt{IsOverloaded}(b, n) \mapsto \bot$
$\texttt{IsOverridden}(s, n) \mapsto \bot$
$\texttt{IsOverridden}(a, n) \mapsto \texttt{IsOverridden}(a, m)$
$\texttt{IsOverridden}(b, n) \mapsto \texttt{IsOverridden}(b, m)$

$\texttt{IsOverriding}(a, n) \mapsto \texttt{IsOverriding}(a, m)$
$\texttt{IsOverriding}(b, n) \mapsto \texttt{IsOverriding}(b, m)$
$\texttt{IsRecursiveMethod}(s, n) \mapsto \bot$
$\texttt{IsRecursiveMethod}(a, n) \mapsto \bot$
$\texttt{IsRecursiveMethod}(b, n) \mapsto \bot$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(s, n, N, V) \mapsto$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(s, m, N, V)$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(a, n, N, V) \mapsto$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(a, m, N, V)$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(b, n, N, V) \mapsto$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(b, m, N, V)$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(s, n, N) \mapsto$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(s, m, N)$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(a, n, N) \mapsto$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(a, m, N)$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(b, n, N) \mapsto$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(b, m, N)$
$\texttt{IsPrivate}(s, V) \mapsto \bot$
$\texttt{IsPrivate}(a, V) \mapsto \bot$
$\texttt{IsPrivate}(b, V) \mapsto \bot$
$\texttt{IsPrivate}(s, n) \mapsto \bot$
$\texttt{IsPrivate}(a, n) \mapsto \bot$
$\texttt{IsPrivate}(b, n) \mapsto \bot$
$\texttt{IsOverriding}(a, n) \mapsto \texttt{IsOverriding}(a, m)$
$\texttt{IsOverriding}(b, n) \mapsto \texttt{IsOverriding}(b, m)$
$\texttt{IsDelegator}(a, n, V) \mapsto \texttt{IsDelegator}(a, m, V)$
$\texttt{IsDelegator}(b, n, V) \mapsto \texttt{IsDelegator}(b, m, V)$
$\texttt{IsDelegator}(a, V, n) \mapsto \texttt{IsDelegator}(a, V, m)$
$\texttt{IsDelegator}(b, V, n) \mapsto \texttt{IsDelegator}(b, V, m)$
$\texttt{IsInheritedMethodWithParams}(a, n, [t; t']) \mapsto \texttt{IsVisibleMethod}(s, m, [t; t'], a)$
$\texttt{IsInheritedMethodWithParams}(b, n, [t; t']) \mapsto \texttt{IsVisibleMethod}(s, m, [t; t'], b)$
$\texttt{IsVisibleMethod}(s, m, [t; t'], a) \mapsto \top$
$\texttt{IsVisibleMethod}(s, m, [t; t'], b) \mapsto \top$
$\texttt{MethodIsUsedWithType}(a, n, [t; t'], [t; t']) \mapsto \texttt{MethodIsUsedWithType}(a, m, [t; t'], [t; t'])$
$\texttt{MethodIsUsedWithType}(b, n, [t; t'], [t; t']) \mapsto \texttt{MethodIsUsedWithType}(b, m, [t; t'], [t; t'])$
$\texttt{IsUsedMethod}(a, n, [t; t']) \mapsto \texttt{IsUsedMethod}(a, m, [t; t'])$
$\texttt{IsUsedMethod}(b, n, [t; t']) \mapsto \texttt{IsUsedMethod}(b, m, [t; t'])$
$\texttt{IsUsedMethodIn}(a, n, V) \mapsto \texttt{IsUsedMethodIn}(a, m, V)$
$\texttt{IsUsedMethodIn}(b, n, V) \mapsto \texttt{IsUsedMethodIn}(b, m, V)$
$\texttt{IsInverter}(a, n, V, V1) \mapsto \texttt{IsInverter}(a, m, V, V1)$
$\texttt{IsInverter}(b, n, V, V1) \mapsto \texttt{IsInverter}(b, m, V, V1)$
$\texttt{ExistsMethodInvocation}(a, V, V1, n) \mapsto \texttt{ExistsMethodInvocation}(a, V, V1, m)$
$\texttt{ExistsMethodInvocation}(b, V, V1, n) \mapsto \texttt{ExistsMethodInvocation}(b, V, V1, m)$
$\texttt{IsIndirectlyRecursive}(a, n) \mapsto \texttt{IsIndirectlyRecursive}(a, m)$
$\texttt{IsIndirectlyRecursive}(b, n) \mapsto \texttt{IsIndirectlyRecursive}(b, m)$
$\texttt{BoundVariableInMethodBody}(a, n, V) \mapsto \texttt{BoundVariableInMethodBody}(a, n, V)$
$\texttt{BoundVariableInMethodBody}(b, n, V) \mapsto \texttt{BoundVariableInMethodBody}(b, n, V)$

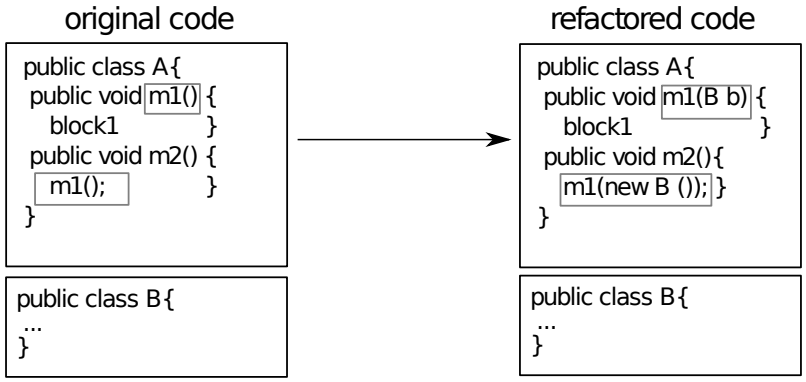
$\texttt{IsOverridden}(a, n) \mapsto \texttt{IsOverridden}(a, m)$
$\texttt{IsOverridden}(b, n) \mapsto \texttt{IsOverridden}(b, m)$

3 AddParameter

(*Add Parameter* in Fowler [Fow99] et [Koc02])

AddParameter(class c, method m, parameterType t, parameterName n, **defaultvalue** e) :

Ajouter un paramètre de type t pour la méthode m dans la classe c. Dans les invocations de la méthode, utiliser l'expression e comme un nouveau paramètre.



Outils de refactoring. *Change Method signature* avec Eclipse tool et *Change Signature* avec IntelliJ IDEA.

4 AddParameterWithReuse

Rôle. AddParameterWithReuse (classname s,subclasses [a,b], methodname m,methodparameters [] ,p ajouter le paramètre p de type t aux paramètres des méthodes s : :m, a : :m et b : :m. Même chose pour AddParameter, mais au lieu d'ajouter une valeur par défaut pour le paramètre ajouté aux invocations, utiliser n'importe quelle valeur avec le type spécifié qui est visible à partir du site de l'invocation.

Avec IntelliJ IDEA, ceci est effectué à l'aide de l'option *Any Var* de l'opération *Change Signature*. Ceci n'est pas disponible avec Eclipse.

Précondition.
$(\neg\texttt{BoundVariableInMethodBody}(s, m, p)$
$\wedge\ \texttt{ExistsClass}(s)$
$\wedge\ \texttt{ExistsMethodDefinition}(s, m)$
$\wedge\ \texttt{ExistsMethodDefinitionWithParams}(s, m, [\,])$
$\wedge\ \neg\texttt{ExistsMethodDefinitionWithParams}(s, m, [t])$
$\wedge\ \neg\texttt{IsInheritedMethodWithParams}(s, m, [t])$
$\wedge\ \neg\texttt{ExistsParameterWithName}(s, m, [\,], p)$
$\wedge\ \texttt{ExistsType}(t)$
$\wedge\ \texttt{AllSubclasses}(s, [a; b]))$

Rétro-description. $\texttt{ExistsMethodDefinitionWithParams}(s, m, [\,]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(s, m, [t]) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(a, m, [\,]) \mapsto \neg\texttt{ExistsMethodDefinitionWithParams}(a, m, [\,])$
$\texttt{ExistsMethodDefinitionWithParams}(b, m, [\,]) \mapsto \neg\texttt{ExistsMethodDefinitionWithParams}(b, m, [\,])$

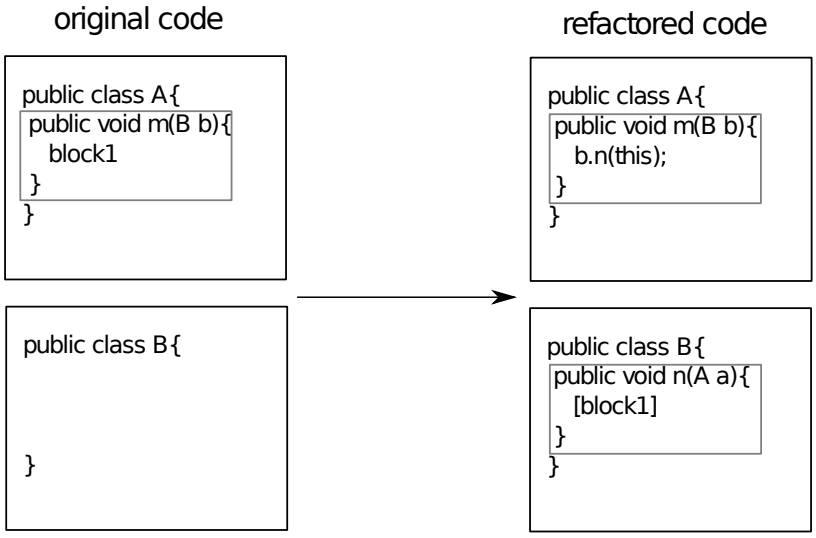
$\texttt{ExistsMethodDefinitionWithParams}(a, m, [t]) \mapsto \texttt{ExistsMethodDefinitionWithParams}(a, m, [\,])$
$\texttt{ExistsMethodDefinitionWithParams}(b, m, [t]) \mapsto \texttt{ExistsMethodDefinitionWithParams}(b, m, [\,])$
$\texttt{ExistsParameterWithName}(s, m, [t], p) \mapsto \top$
$\texttt{ExistsParameterWithName}(a, m, [t], p) \mapsto \top$
$\texttt{ExistsParameterWithName}(b, m, [t], p) \mapsto \top$
$\texttt{ExistsParameterWithType}(s, m, [t], t) \mapsto \top$
$\texttt{ExistsParameterWithType}(a, m, [t], t) \mapsto \top$
$\texttt{ExistsParameterWithType}(b, m, [t], t) \mapsto \top$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(s, m, p, T) \mapsto \top$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(a, m, p, T) \mapsto \top$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(b, m, p, T) \mapsto \top$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(s, m, p) \mapsto$
$(\neg\texttt{IsOverloaded}(s, m)$
$\wedge \neg\texttt{IsOverloaded}(a, m)$
$\wedge \neg\texttt{IsOverloaded}(b, m))$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(a, m, p) \mapsto$
$(\neg\texttt{IsOverloaded}(s, m)$
$\wedge \neg\texttt{IsOverloaded}(a, m)$
$\wedge \neg\texttt{IsOverloaded}(b, m))$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(b, m, p) \mapsto$
$(\neg\texttt{IsOverloaded}(s, m)$
$\wedge \neg\texttt{IsOverloaded}(a, m)$
$\wedge \neg\texttt{IsOverloaded}(b, m))$
$\texttt{IsUsedConstructorAsMethodParameter}(t, s, m) \mapsto \top$
$\texttt{IsUsedConstructorAsMethodParameter}(t, a, m) \mapsto \top$
$\texttt{IsUsedConstructorAsMethodParameter}(t, b, m) \mapsto \top$

5 MoveMethodWithDelegate

(*Move Method* in Fowler [Fow99])

Rôle. MoveMethodWithDelegate (classname s,attributes [att1,att2], targetclass a,methodtobemoved m, parameterstypes [t,a], returntype r, movedmethod n, receivingobjectname o, newreceivingobjectname o') : déplacer la méthode s : :m vers la classe a et la renommer en n.

Transformer la méthode s : :m en un délégateur pour la méthode a : :n. Le code de la méthode m est déplacée vers n.



Outils de refactoring. *Move* avec Eclipse tool. Avec IntelliJ IDEA, introduire tout d'abord un délégateur local, puis utiliser *Move*.

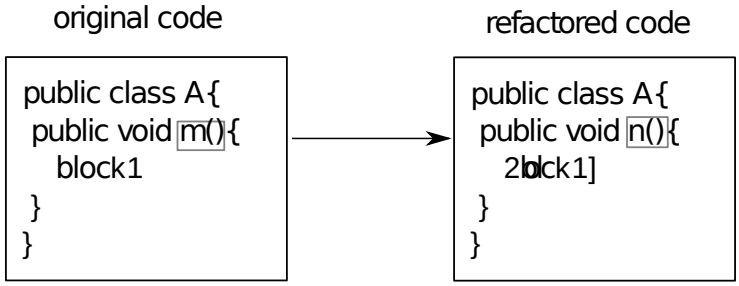
Précondition.
$(\texttt{ExistsClass}(s)$
$\wedge\, \texttt{ExistsClass}(a)$
$\wedge\, \texttt{ExistsMethodDefinitionWithParams}(s, m, [t; a])$
$\wedge\, \texttt{ExistsParameterWithType}(s, m, [t; a], a)$
$\wedge\, \texttt{ExistsParameterWithName}(s, m, [t; a], o)$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(a, n, [t; s])$
$\wedge\, \texttt{HasReturnType}(s, m, r)$
$\wedge\, \neg\texttt{IsPrivate}(s, m)$
$\wedge\, \neg\texttt{IsPrivate}(s, att1)$
$\wedge\, \neg\texttt{IsPrivate}(s, att2))$

Rétro-description.
$\texttt{ExistsMethodDefinitionWithParams}(s, m, [t; a]) \mapsto \top$
$\texttt{ExistsMethodDefinition}(a, n) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(a, n, [t; s]) \mapsto \top$
$\texttt{HasReturnType}(a, n, r) \mapsto \texttt{HasReturnType}(s, m, r)$
$\texttt{BoundVariableInMethodBody}(a, n, M) \mapsto \texttt{BoundVariableInMethodBody}(s, m, M)$
$\texttt{ExistsParameterWithName}(a, n, [t; s], N) \mapsto \texttt{ExistsParameterWithName}(s, m, [t; a], N)(condition)$
$\texttt{ExistsParameterWithName}(a, n, [t; s], o') \mapsto \top$
$\texttt{ExistsParameterWithName}(a, n, [t; s], o) \mapsto \bot$
$\texttt{ExistsParameterWithType}(a, n, [t; s], s) \mapsto \top$
$\texttt{ExistsParameterWithType}(a, n, [t; s], a) \mapsto \bot$
$\texttt{ExistsParameterWithType}(a, n, [t; s], T) \mapsto \texttt{ExistsParameterWithType}(s, m, [t; a], T)(condition)$
$\texttt{ExistsMethodInvocation}(s, m, a, n) \mapsto \top$
$\texttt{IsInverter}(s, m, a, r) \mapsto \top$
$\texttt{IsPrivate}(s, att1) \mapsto \bot$
$\texttt{IsPrivate}(s, att2) \mapsto \bot$
$\texttt{IsUsedAttributeInMethodBody}(s, att1, m) \mapsto \bot$
$\texttt{IsUsedAttributeInMethodBody}(s, att2, m) \mapsto \bot$

6 RenameMethod

(*Rename* in Fowler [Fow99] et [Koc02])

RenameMethod(class c, method m, newname n) : Renommer la méthode c : :m en n.



Outils de refactoring. *Rename* avec Eclipse and IntelliJ IDEA.

Nous avons identifié deux types d'utilisation de cette opération. Une utilisation qui n'accepte pas la surcharge et une autre qui accepte la surcharge.

7 RenameInHierarchyNoOverloading

Rôle RenameInHierarchyNoOverloading (class c,subclasses [a,b],method m ,types [t,t'], newname n) : renommer la méthode (c,a,b) : :m en n si n n'existe pas déjà avec une autre signature dans la hiérarchie.

Précondition.

$(\texttt{ExistsClass}(c)$
$\wedge\, \texttt{ExistsClass}(a)$
$\wedge\, \texttt{ExistsClass}(b)$
$\wedge\, \texttt{ExistsMethodDefinition}(c, m)$
$\wedge\, \texttt{ExistsMethodDefinitionWithParams}(c, m, [t; t'])$
$\wedge\, \texttt{AllSubclasses}(c, [a; b])$
$\wedge\, \neg\texttt{ExistsMethodDefinition}(c, n)$
$\wedge\, \neg\texttt{ExistsMethodDefinition}(a, n)$
$\wedge\, \neg\texttt{ExistsMethodDefinition}(b, n)$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(c, n, [t; t'])$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(a, n, [t; t'])$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(b, n, [t; t'])$
$\wedge\, \neg\texttt{IsOverloaded}(c, m)$
$\wedge\, \neg\texttt{IsOverloaded}(a, m)$
$\wedge\, \neg\texttt{IsOverloaded}(b, m)$
$\wedge\, \neg\texttt{IsInheritedMethod}(c, n))$

Rétro-description.

$\texttt{ExistsMethodDefinition}(c, n) \mapsto \top$

$\texttt{ExistsMethodDefinitionWithParams}(c, n, [t; t']) \mapsto \top$

$\texttt{ExistsMethodDefinition}(a, n) \mapsto \texttt{ExistsMethodDefinition}(a, m)$

$\texttt{ExistsMethodDefinition}(b, n) \mapsto \texttt{ExistsMethodDefinition}(b, m)$

$\texttt{ExistsMethodDefinition}(c, m) \mapsto \bot$
$\texttt{ExistsMethodDefinition}(a, m) \mapsto \bot$
$\texttt{ExistsMethodDefinition}(b, m) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(a, n, [t; t']) \mapsto \texttt{ExistsMethodDefinitionWithParams}(a, m, [t; t'])$
$\texttt{ExistsMethodDefinitionWithParams}(b, n, [t; t']) \mapsto \texttt{ExistsMethodDefinitionWithParams}(b, m, [t; t'])$
$\texttt{ExistsMethodDefinitionWithParams}(c, m, [t; t']) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(a, m, [t; t']) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(b, m, [t; t']) \mapsto \bot$
$\texttt{IsInheritedMethod}(a, n) \mapsto \texttt{IsInheritedMethod}(a, m)$
$\texttt{IsInheritedMethod}(b, n) \mapsto \texttt{IsInheritedMethod}(b, m)$
$\texttt{IsDelegator}(c, n, V) \mapsto \texttt{IsDelegator}(c, m, V)$
$\texttt{IsDelegator}(a, n, V) \mapsto \texttt{IsDelegator}(a, m, V)$
$\texttt{IsDelegator}(b, n, V) \mapsto \texttt{IsDelegator}(b, m, V)$
$\texttt{IsDelegator}(c, V, n) \mapsto \texttt{IsDelegator}(c, V, m)$
$\texttt{IsDelegator}(a, V, n) \mapsto \texttt{IsDelegator}(a, V, m)$
$\texttt{IsDelegator}(b, V, n) \mapsto \texttt{IsDelegator}(b, V, m)$
$\texttt{IsOverloaded}(c, V) \mapsto \texttt{IsOverloaded}(c, V)(condition)$
$\texttt{IsOverloaded}(a, V) \mapsto \texttt{IsOverloaded}(a, V)(condition)$
$\texttt{IsOverloaded}(b, V) \mapsto \texttt{IsOverloaded}(b, V)(condition)$
$\texttt{IsOverriding}(a, n) \mapsto \texttt{IsOverriding}(a, m)$
$\texttt{IsOverriding}(b, n) \mapsto \texttt{IsOverriding}(b, m)$
$\texttt{IsOverridden}(a, n) \mapsto \texttt{IsOverridden}(a, m)$
$\texttt{IsOverridden}(b, n) \mapsto \texttt{IsOverridden}(b, m)$
$\texttt{ExistsParameterWithName}(c, n, [t; t'], V1) \mapsto \texttt{ExistsParameterWithName}(c, m, [t; t'], V1)$
$\texttt{ExistsParameterWithName}(a, n, [t; t'], V1) \mapsto \texttt{ExistsParameterWithName}(a, m, [t; t'], V1)$
$\texttt{ExistsParameterWithName}(b, n, [t; t'], V1) \mapsto \texttt{ExistsParameterWithName}(b, m, [t; t'], V1)$
$\texttt{ExistsParameterWithType}(c, n, [t; t'], V1) \mapsto \texttt{ExistsParameterWithType}(c, m, [t; t'], V1)$
$\texttt{ExistsParameterWithType}(a, n, [t; t'], V1) \mapsto \texttt{ExistsParameterWithType}(a, m, [t; t'], V1)$
$\texttt{ExistsParameterWithType}(b, n, [t; t'], V1) \mapsto \texttt{ExistsParameterWithType}(b, m, [t; t'], V1)$
$\texttt{IsRecursiveMethod}(c, n) \mapsto \texttt{IsRecursiveMethod}(c, m)$
$\texttt{IsRecursiveMethod}(a, n) \mapsto \texttt{IsRecursiveMethod}(a, m)$
$\texttt{IsRecursiveMethod}(b, n) \mapsto \texttt{IsRecursiveMethod}(b, m)$
$\texttt{ExistsAbstractMethod}(c, n) \mapsto \texttt{ExistsAbstractMethod}(c, m)$
$\texttt{ExistsAbstractMethod}(a, n) \mapsto \texttt{ExistsAbstractMethod}(a, m)$
$\texttt{ExistsAbstractMethod}(b, n) \mapsto \texttt{ExistsAbstractMethod}(b, m)$
$\texttt{IsInheritedMethodWithParams}(a, n, [t; t']) \mapsto \texttt{IsVisibleMethod}(c, m, [t; t'], a)$
$\texttt{IsInheritedMethodWithParams}(b, n, [t; t']) \mapsto \texttt{IsVisibleMethod}(c, m, [t; t'], b)$
$\texttt{MethodIsUsedWithType}(c, n, [t; t'], [t; t']) \mapsto \texttt{MethodIsUsedWithType}(c, m, [t; t'], [t; t'])$
$\texttt{MethodIsUsedWithType}(a, n, [t; t'], [t; t']) \mapsto \texttt{MethodIsUsedWithType}(a, m, [t; t'], [t; t'])$
$\texttt{MethodIsUsedWithType}(b, n, [t; t'], [t; t']) \mapsto \texttt{MethodIsUsedWithType}(b, m, [t; t'], [t; t'])$
$\texttt{IsUsedMethod}(c, n, [t; t']) \mapsto \texttt{IsUsedMethod}(c, m, [t; t'])$
$\texttt{IsUsedMethod}(a, n, [t; t']) \mapsto \texttt{IsUsedMethod}(a, m, [t; t'])$
$\texttt{IsUsedMethod}(b, n, [t; t']) \mapsto \texttt{IsUsedMethod}(b, m, [t; t'])$
$\texttt{IsUsedMethodIn}(c, n, V) \mapsto \texttt{IsUsedMethodIn}(c, m, V)$
$\texttt{IsUsedMethodIn}(a, n, V) \mapsto \texttt{IsUsedMethodIn}(a, m, V)$
$\texttt{IsUsedMethodIn}(b, n, V) \mapsto \texttt{IsUsedMethodIn}(b, m, V)$

$\texttt{HasReturnType}(c, n, V1) \mapsto \texttt{HasReturnType}(c, m, V1)$
$\texttt{HasReturnType}(a, n, V1) \mapsto \texttt{HasReturnType}(a, m, V1)$
$\texttt{HasReturnType}(b, n, V1) \mapsto \texttt{HasReturnType}(b, m, V1)$
$\texttt{IsInverter}(c, n, V, V1) \mapsto \texttt{IsInverter}(c, m, V, V1)$
$\texttt{IsInverter}(a, n, V, V1) \mapsto \texttt{IsInverter}(a, m, V, V1)$
$\texttt{IsInverter}(b, n, V, V1) \mapsto \texttt{IsInverter}(b, m, V, V1)$
$\texttt{ExistsMethodInvocation}(c, V, V1, n) \mapsto \texttt{ExistsMethodInvocation}(c, V, V1, m)$
$\texttt{ExistsMethodInvocation}(a, V, V1, n) \mapsto \texttt{ExistsMethodInvocation}(a, V, V1, m)$
$\texttt{ExistsMethodInvocation}(b, V, V1, n) \mapsto \texttt{ExistsMethodInvocation}(b, V, V1, m)$
$\texttt{ExistsMethodInvocation}(c, m, V, V1) \mapsto \bot$
$\texttt{ExistsMethodInvocation}(a, m, V, V1) \mapsto \bot$
$\texttt{ExistsMethodInvocation}(b, m, V, V1) \mapsto \bot$
$\texttt{IsIndirectlyRecursive}(c, n) \mapsto \texttt{IsIndirectlyRecursive}(c, m)$
$\texttt{IsIndirectlyRecursive}(a, n) \mapsto \texttt{IsIndirectlyRecursive}(a, m)$
$\texttt{IsIndirectlyRecursive}(b, n) \mapsto \texttt{IsIndirectlyRecursive}(b, m)$
$\texttt{BoundVariableInMethodBody}(c, n, V) \mapsto \texttt{BoundVariableInMethodBody}(c, n, V)$
$\texttt{BoundVariableInMethodBody}(a, n, V) \mapsto \texttt{BoundVariableInMethodBody}(a, n, V)$
$\texttt{BoundVariableInMethodBody}(b, n, V) \mapsto \texttt{BoundVariableInMethodBody}(b, n, V)$
$\texttt{IsOverridden}(a, n) \mapsto \texttt{IsOverridden}(a, m)$
$\texttt{IsOverridden}(b, n) \mapsto \texttt{IsOverridden}(b, m)$
$\texttt{IsUsedConstructorAsMethodParameter}(V, c, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(V, a, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(V, b, m) \mapsto \bot$

8 RenameOverloadedMethodInHierarchy

Rôle RenameOverloadedMethodInHierarchy (class c,subclasses [a,b],method m ,usedconstrcutorsInM [c1,c2], newname n, types [t]) : renommer la méthode (c,a,b) : :m en n peu n'importe si n est surchargée ou non.

Précondition.
$(\texttt{ExistsClass}(c)$
$\wedge\, \texttt{ExistsMethodDefinitionWithParams}(c, m, [t])$
$\wedge\, \neg\texttt{IsInheritedMethodWithParams}(c, n, [t])$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(c, n, [t])$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(a, n, [t])$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(b, n, [t])$
$\wedge\, \neg\texttt{IsInheritedMethodWithParams}(c, m, [t])$
$\wedge\, \texttt{AllSubclasses}(c, [a; b]))$

Rétro-description.
$\texttt{ExistsMethodDefinition}(c, m) \mapsto \bot$
$\texttt{ExistsMethodDefinition}(c, n) \mapsto \top$
$\texttt{IsOverriding}(c, n) \mapsto \bot$
$\texttt{IsOverridden}(c, n) \mapsto \bot$

$\texttt{IsPublic}(c, n) \mapsto \texttt{IsPublic}(c, m)$
$\texttt{ExistsMethodDefinitionWithParams}(c, n, [t]) \mapsto \texttt{ExistsMethodDefinitionWithParams}(c, m, [t])$
$\texttt{ExistsMethodDefinitionWithParams}(a, n, [t]) \mapsto \texttt{ExistsMethodDefinitionWithParams}(a, m, [t])$
$\texttt{ExistsMethodDefinitionWithParams}(b, n, [t]) \mapsto \texttt{ExistsMethodDefinitionWithParams}(b, m, [t])$
$\texttt{ExistsMethodDefinition}(a, n) \mapsto \top$
$\texttt{ExistsMethodDefinition}(b, n) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(c, m, [t]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(a, m, [t]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(b, m, [t]) \mapsto \bot$
$\texttt{ExistsMethodDefinition}(a, m) \mapsto \bot$
$\texttt{ExistsMethodDefinition}(b, m) \mapsto \bot$
$\texttt{isOverridingMethod}(a, n, [t]) \mapsto \texttt{isOverridingMethod}(a, m, [t])$
$\texttt{isOverridingMethod}(b, n, [t]) \mapsto \texttt{isOverridingMethod}(b, m, [t])$
$\texttt{ExistsParameterWithName}(c, n, [t], V) \mapsto \texttt{ExistsParameterWithName}(c, m, [t], V)$
$\texttt{ExistsParameterWithName}(a, n, [t], V) \mapsto \texttt{ExistsParameterWithName}(a, m, [t], V)$
$\texttt{ExistsParameterWithName}(b, n, [t], V) \mapsto \texttt{ExistsParameterWithName}(b, m, [t], V)$
$\texttt{ExistsParameterWithType}(c, n, [t], V) \mapsto \texttt{ExistsParameterWithType}(c, m, [t], V)$
$\texttt{ExistsParameterWithType}(a, n, [t], V) \mapsto \texttt{ExistsParameterWithType}(a, m, [t], V)$
$\texttt{ExistsParameterWithType}(b, n, [t], V) \mapsto \texttt{ExistsParameterWithType}(b, m, [t], V)$
$\texttt{IsDelegator}(c, n, V) \mapsto \texttt{IsDelegator}(c, m, V)$
$\texttt{IsDelegator}(a, n, V) \mapsto \texttt{IsDelegator}(a, m, V)$
$\texttt{IsDelegator}(b, n, V) \mapsto \texttt{IsDelegator}(b, m, V)$
$\texttt{IsDelegator}(c, V, n) \mapsto \texttt{IsDelegator}(c, V, m)$
$\texttt{IsDelegator}(a, V, n) \mapsto \texttt{IsDelegator}(a, V, m)$
$\texttt{IsDelegator}(b, V, n) \mapsto \texttt{IsDelegator}(b, V, m)$
$\texttt{IsRecursiveMethod}(c, n) \mapsto \texttt{IsRecursiveMethod}(c, m)$
$\texttt{IsRecursiveMethod}(a, n) \mapsto \texttt{IsRecursiveMethod}(a, m)$
$\texttt{IsRecursiveMethod}(b, n) \mapsto \texttt{IsRecursiveMethod}(b, m)$
$\texttt{ExistsAbstractMethod}(c, n) \mapsto \texttt{ExistsAbstractMethod}(c, m)$
$\texttt{ExistsAbstractMethod}(a, n) \mapsto \texttt{ExistsAbstractMethod}(a, m)$
$\texttt{ExistsAbstractMethod}(b, n) \mapsto \texttt{ExistsAbstractMethod}(b, m)$
$\texttt{IsInheritedMethodWithParams}(a, n, [t]) \mapsto \texttt{IsVisibleMethod}(c, m, [t], a)$
$\texttt{IsInheritedMethodWithParams}(b, n, [t]) \mapsto \texttt{IsVisibleMethod}(c, m, [t], b)$
$\texttt{MethodIsUsedWithType}(c, n, [t], [t]) \mapsto \texttt{MethodIsUsedWithType}(c, m, [t], [t])$
$\texttt{MethodIsUsedWithType}(a, n, [t], [t]) \mapsto \texttt{MethodIsUsedWithType}(a, m, [t], [t])$
$\texttt{MethodIsUsedWithType}(b, n, [t], [t]) \mapsto \texttt{MethodIsUsedWithType}(b, m, [t], [t])$
$\texttt{IsUsedMethod}(c, n, [t]) \mapsto \texttt{IsUsedMethod}(c, m, [t])$
$\texttt{IsUsedMethod}(a, n, [t]) \mapsto \texttt{IsUsedMethod}(a, m, [t])$
$\texttt{IsUsedMethod}(b, n, [t]) \mapsto \texttt{IsUsedMethod}(b, m, [t])$
$\texttt{IsUsedMethodIn}(c, n, V) \mapsto \texttt{IsUsedMethodIn}(c, m, V)$
$\texttt{IsUsedMethodIn}(a, n, V) \mapsto \texttt{IsUsedMethodIn}(a, m, V)$
$\texttt{IsUsedMethodIn}(b, n, V) \mapsto \texttt{IsUsedMethodIn}(b, m, V)$
$\texttt{HasReturnType}(c, n, V) \mapsto \texttt{HasReturnType}(c, m, V)$
$\texttt{HasReturnType}(a, n, V) \mapsto \texttt{HasReturnType}(a, m, V)$
$\texttt{HasReturnType}(b, n, V) \mapsto \texttt{HasReturnType}(b, m, V)$
$\texttt{IsInverter}(c, n, V, V1) \mapsto \texttt{IsInverter}(c, m, V, V1)$

$\texttt{IsInverter}(a, n, V, V1) \mapsto \texttt{IsInverter}(a, m, V, V1)$
$\texttt{IsInverter}(b, n, V, V1) \mapsto \texttt{IsInverter}(b, m, V, V1)$
$\texttt{ExistsMethodInvocation}(c, V, V1, n) \mapsto \texttt{ExistsMethodInvocation}(c, V, V1, m)$
$\texttt{ExistsMethodInvocation}(a, V, V1, n) \mapsto \texttt{ExistsMethodInvocation}(a, V, V1, m)$
$\texttt{ExistsMethodInvocation}(b, V, V1, n) \mapsto \texttt{ExistsMethodInvocation}(b, V, V1, m)$
$\texttt{IsIndirectlyRecursive}(c, n) \mapsto \texttt{IsIndirectlyRecursive}(c, m)$
$\texttt{IsIndirectlyRecursive}(a, n) \mapsto \texttt{IsIndirectlyRecursive}(a, m)$
$\texttt{IsIndirectlyRecursive}(b, n) \mapsto \texttt{IsIndirectlyRecursive}(b, m)$
$\texttt{BoundVariableInMethodBody}(c, n, V) \mapsto \texttt{BoundVariableInMethodBody}(c, m, V)$
$\texttt{BoundVariableInMethodBody}(a, n, V) \mapsto \texttt{BoundVariableInMethodBody}(a, m, V)$
$\texttt{BoundVariableInMethodBody}(b, n, V) \mapsto \texttt{BoundVariableInMethodBody}(b, m, V)$
$\texttt{IsOverridden}(a, n) \mapsto \texttt{IsOverridden}(a, m)$
$\texttt{IsOverridden}(b, n) \mapsto \texttt{IsOverridden}(b, m)$
$\texttt{IsUsedConstructorAsMethodParameter}(V, c, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(V, a, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(V, b, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(V, c, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(V, a, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(V, b, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(V, c, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(V, a, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(V, b, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(V, c, n) \mapsto \texttt{IsUsedConstructorAsMethodParameter}(V, c, m)$
$\texttt{IsUsedConstructorAsMethodParameter}(V, a, n) \mapsto \texttt{IsUsedConstructorAsMethodParameter}(V, a, m)$
$\texttt{IsUsedConstructorAsMethodParameter}(V, b, n) \mapsto \texttt{IsUsedConstructorAsMethodParameter}(V, b, m)$
$\texttt{IsUsedConstructorAsObjectReceiver}(c1, c, n) \mapsto \top$
$\texttt{IsUsedConstructorAsObjectReceiver}(c2, c, n) \mapsto \top$
$\texttt{IsUsedConstructorAsObjectReceiver}(c1, a, n) \mapsto \top$
$\texttt{IsUsedConstructorAsObjectReceiver}(c2, a, n) \mapsto \top$
$\texttt{IsUsedConstructorAsObjectReceiver}(c1, b, n) \mapsto \top$
$\texttt{IsUsedConstructorAsObjectReceiver}(c2, b, n) \mapsto \top$
$\texttt{IsUsedConstructorAsMethodParameter}(V, c, n) \mapsto \texttt{IsUsedConstructorAsMethodParameter}(V, c, m)$
$\texttt{IsUsedConstructorAsMethodParameter}(V, a, n) \mapsto \texttt{IsUsedConstructorAsMethodParameter}(V, a, m)$
$\texttt{IsUsedConstructorAsMethodParameter}(V, b, n) \mapsto \texttt{IsUsedConstructorAsMethodParameter}(V, b, m)$

9 RenameDelegatorWithOverloading

Rôle RenameDelegatorWithOverloading (classname s, subclasses [a,b], method m,paramtype t,paramName pn,super typeOfparamtype t', newname n) : renommer la méthode (c,a,b) : :m en n avec l'acceptation de la surcharge de la méthode n. Cette opération est une utilisation ad-hoc de l'opération RenameOverloadedMethodInHierarchy (nous avons besoin de plus d'informations sur la signature de la méthode à renommer).

Précondition.
$(\texttt{ExistsClass}(s)$
$\wedge \texttt{ExistsClass}(a)$

$\wedge$ ExistsClass(b)
$\wedge$ ExistsMethodDefinition(s, m)
$\wedge$ ExistsMethodDefinitionWithParams$(s, m, [t])$
$\wedge$ AllSubclasses$(s, [a; b])$
$\wedge \neg$ExistsMethodDefinitionWithParams$(s, n, [t])$
$\wedge \neg$ExistsMethodDefinitionWithParams$(a, n, [t])$
$\wedge \neg$ExistsMethodDefinitionWithParams$(b, n, [t])$
$\wedge \neg$IsInheritedMethod(s, n)
$\wedge \neg$ExistsMethodDefinitionWithParams$(s, n, [t]))$

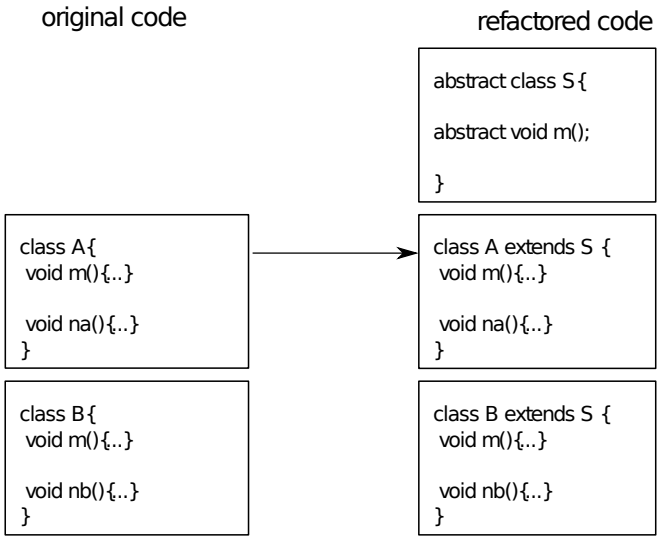
Rétro-description.
ExistsMethodDefinition$(s, n) \mapsto \top$
ExistsMethodDefinitionWithParams$(s, n, [t]) \mapsto \top$
IsPublic$(s, n) \mapsto$ IsPublic(s, m)
ExistsMethodDefinition$(s, m) \mapsto \bot$
ExistsMethodDefinition$(a, m) \mapsto \bot$
ExistsMethodDefinition$(b, m) \mapsto \bot$
ExistsMethodDefinitionWithParams$(s, m, [t]) \mapsto \bot$
ExistsMethodDefinitionWithParams$(a, m, [t]) \mapsto \bot$
ExistsMethodDefinitionWithParams$(b, m, [t]) \mapsto \bot$
ExistsMethodDefinition$(a, n) \mapsto$ ExistsMethodDefinition(a, m)
ExistsMethodDefinition$(b, n) \mapsto$ ExistsMethodDefinition(b, m)
ExistsMethodDefinitionWithParams$(a, n, [t]) \mapsto$ ExistsMethodDefinitionWithParams$(a, m, [t])$
ExistsMethodDefinitionWithParams$(b, n, [t]) \mapsto$ ExistsMethodDefinitionWithParams$(b, m, [t])$
IsInheritedMethod$(a, n) \mapsto$ IsInheritedMethod(a, m)
IsInheritedMethod$(b, n) \mapsto$ IsInheritedMethod(b, m)
MethodIsUsedWithType$(s, n, [t], [t]) \mapsto$ MethodIsUsedWithType$(s, m, [t], [t])$
MethodIsUsedWithType$(a, n, [t], [t]) \mapsto$ MethodIsUsedWithType$(a, m, [t], [t])$
MethodIsUsedWithType$(b, n, [t], [t]) \mapsto$ MethodIsUsedWithType$(b, m, [t], [t])$
MethodIsUsedWithType$(s, m, [t], [t]) \mapsto \bot$
MethodIsUsedWithType$(a, m, [t], [t]) \mapsto \bot$
MethodIsUsedWithType$(b, m, [t], [t]) \mapsto \bot$
ExistsParameterWithName$(s, n, [t], V) \mapsto$ ExistsParameterWithName$(s, m, [t], V)$
ExistsParameterWithName$(a, n, [t], V) \mapsto$ ExistsParameterWithName$(a, m, [t], V)$
ExistsParameterWithName$(b, n, [t], V) \mapsto$ ExistsParameterWithName$(b, m, [t], V)$
ExistsParameterWithType$(s, n, [t], V) \mapsto$ ExistsParameterWithType$(s, m, [t], V)$
ExistsParameterWithType$(a, n, [t], V) \mapsto$ ExistsParameterWithType$(a, m, [t], V)$
ExistsParameterWithType$(b, n, [t], V) \mapsto$ ExistsParameterWithType$(b, m, [t], V)$
ExistsMethodInvocation$(s, V1, V, n) \mapsto$ ExistsMethodInvocation$(s, V1, V, m)$
ExistsMethodInvocation$(a, V1, V, n) \mapsto$ ExistsMethodInvocation$(a, V1, V, m)$
ExistsMethodInvocation$(b, V1, V, n) \mapsto$ ExistsMethodInvocation$(b, V1, V, m)$
IsDelegator$(s, n, V) \mapsto$ IsDelegator(s, m, V)
IsDelegator$(a, n, V) \mapsto$ IsDelegator(a, m, V)
IsDelegator$(b, n, V) \mapsto$ IsDelegator(b, m, V)
IsDelegator$(s, V, n) \mapsto$ IsDelegator(s, V, m)
IsDelegator$(a, V, n) \mapsto$ IsDelegator(a, V, m)

$\texttt{IsDelegator}(b, V, n) \mapsto \texttt{IsDelegator}(b, V, m)$
$\texttt{IsUsedMethod}(s, n, [t]) \mapsto \texttt{IsUsedMethod}(s, m, [t])$
$\texttt{IsUsedMethod}(a, n, [t]) \mapsto \texttt{IsUsedMethod}(a, m, [t])$
$\texttt{IsUsedMethod}(b, n, [t]) \mapsto \texttt{IsUsedMethod}(b, m, [t])$
$\texttt{IsUsedMethodIn}(s, n, V) \mapsto \texttt{IsUsedMethodIn}(s, m, V)$
$\texttt{IsUsedMethodIn}(a, n, V) \mapsto \texttt{IsUsedMethodIn}(a, m, V)$
$\texttt{IsUsedMethodIn}(b, n, V) \mapsto \texttt{IsUsedMethodIn}(b, m, V)$
$\texttt{HasReturnType}(s, n, V) \mapsto \texttt{HasReturnType}(s, m, V)$
$\texttt{HasReturnType}(a, n, V) \mapsto \texttt{HasReturnType}(a, m, V)$
$\texttt{HasReturnType}(b, n, V) \mapsto \texttt{HasReturnType}(b, m, V)$
$\texttt{IsInverter}(s, n, V, V1) \mapsto \texttt{IsInverter}(s, m, V, V1)$
$\texttt{IsInverter}(a, n, V, V1) \mapsto \texttt{IsInverter}(a, m, V, V1)$
$\texttt{IsInverter}(b, n, V, V1) \mapsto \texttt{IsInverter}(b, m, V, V1)$
$\texttt{IsIndirectlyRecursive}(s, n) \mapsto \texttt{IsIndirectlyRecursive}(s, m)$
$\texttt{IsIndirectlyRecursive}(a, n) \mapsto \texttt{IsIndirectlyRecursive}(a, m)$
$\texttt{IsIndirectlyRecursive}(b, n) \mapsto \texttt{IsIndirectlyRecursive}(b, m)$
$\texttt{BoundVariableInMethodBody}(s, n, V) \mapsto \texttt{BoundVariableInMethodBody}(s, n, V)$
$\texttt{BoundVariableInMethodBody}(a, n, V) \mapsto \texttt{BoundVariableInMethodBody}(a, n, V)$
$\texttt{BoundVariableInMethodBody}(b, n, V) \mapsto \texttt{BoundVariableInMethodBody}(b, n, V)$
$\texttt{IsOverridden}(a, n) \mapsto \texttt{IsOverridden}(a, m)$
$\texttt{IsOverridden}(b, n) \mapsto \texttt{IsOverridden}(b, m)$
$\texttt{IsUsedConstructorAsMethodParameter}(V, s, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(V, a, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(V, b, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(t, s, n) \mapsto \texttt{IsUsedConstructorAsObjectReceiver}(t, s, m)$
$\texttt{IsUsedConstructorAsObjectReceiver}(t, a, n) \mapsto \texttt{IsUsedConstructorAsObjectReceiver}(t, a, m)$
$\texttt{IsUsedConstructorAsObjectReceiver}(t, b, n) \mapsto \texttt{IsUsedConstructorAsObjectReceiver}(t, b, m)$

10 ExtractSuperClass

(*Extract Super Class* in Fowler [Fow99] and [Koc02])

Rôle. ExtractSuperClass (subclasses[a,b], superclass s,methodsOfsubclasses [m,n],returntype t) : extraire une super classe s à partir des classes a et b et déclarer les méthodes a : :m, a : :n, b : :m and b : :n comme méthodes abstraites dans la classe s.



Outils de refactoring. *Extract Superclass* avec Eclipse tool et IntelliJ IDEA. Avec IntelliJ IDEA, l'opération *Extract Superclass* ne peut pas être appliquée pour plusieurs classes à la fois ce qui nous a amené à l'étendre pour l'appliquer sur plusieurs classes.

Précondition.
$(\neg\texttt{ExistsType}(s)$
$\wedge\, \texttt{ExistsClass}(a)$
$\wedge\, \texttt{ExistsClass}(b)$
$\wedge\, \texttt{ExtendsDirectly}(a, java.lang.Object)$
$\wedge\, \texttt{ExtendsDirectly}(b, java.lang.Object)$
$\wedge\, \texttt{HasReturnType}(a, m, t)$
$\wedge\, \texttt{HasReturnType}(a, n, t)$
$\wedge\, \texttt{HasReturnType}(b, m, t)$
$\wedge\, \texttt{HasReturnType}(b, n, t))$

Rétro-description.
$\texttt{IsAbstractClass}(s) \mapsto \top$
$\texttt{ExistsClass}(s) \mapsto \top$
$\texttt{ExistsType}(s) \mapsto \top$
$\texttt{ExistsMethodDefinition}(s, X) \mapsto (\texttt{ExistsMethodDefinition}(a, X)$
$\wedge\, \texttt{ExistsMethodDefinition}(b, X))$
$\texttt{ExistsMethodDefinitionWithParams}(s, X, [\,]) \mapsto (\texttt{ExistsMethodDefinitionWithParams}(a, X, [\,])$
$\wedge\, \texttt{ExistsMethodDefinitionWithParams}(b, X, [\,]))$
$\texttt{ExistsMethodDefinitionWithParams}(s, X, [Y]) \mapsto (\texttt{ExistsMethodDefinitionWithParams}(a, X, [Y])$
$\wedge\, \texttt{ExistsMethodDefinitionWithParams}(b, X, [Y]))$
$\texttt{IsUsedMethodIn}(s, X, Y) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(X, Y, [s]) \mapsto \bot$
$\texttt{IsUsedMethod}(s, X, [Y]) \mapsto \bot$
$\texttt{AllSubclasses}(s, [a; b]) \mapsto \top$
$\texttt{MethodIsUsedWithType}(X, Y, [Z], [s]) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(X, Y, [s]) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(s, X, Y) \mapsto \bot$
$\texttt{IsUsedConstructorAsInitializer}(s, X, Y) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(s, X, Y) \mapsto \bot$

$\texttt{IsUsedConstructorAsMethodParameter}(X, s, Y) \mapsto \bot$
$\texttt{IsUsedConstructorAsInitializer}(X, s, Y) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(X, s, Y) \mapsto \bot$
$\texttt{IsPrimitiveType}(s) \mapsto \bot$
$\texttt{IsSubType}(a, s) \mapsto \top$
$\texttt{IsSubType}(b, s) \mapsto \top$
$\texttt{IsSubType}(X, s) \mapsto \texttt{IsSubType}(X, a)$
$\texttt{IsSubType}(X, s) \mapsto \texttt{IsSubType}(X, b)$
$\texttt{IsPublic}(s, m) \mapsto \top$
$\texttt{IsPublic}(s, n) \mapsto \top$
$\texttt{ExistsAbstractMethod}(s, m) \mapsto \top$
$\texttt{ExistsAbstractMethod}(s, n) \mapsto \top$
$\texttt{IsOverriding}(s, m) \mapsto \bot$
$\texttt{IsOverriding}(s, n) \mapsto \bot$
$\texttt{IsOverridden}(s, m) \mapsto \top$
$\texttt{IsOverridden}(s, n) \mapsto \top$
$\texttt{IsPrivate}(s, m) \mapsto \bot$
$\texttt{IsPrivate}(s, n) \mapsto \bot$

11 ExtractSuperClassWithoutPullUp

Rôle. ExtractSuperClassWithouPullUp (subclasses[a,b], superclass s) : Cette opération est une utilisation spécifique de l'opération *ExtractSuperClass*. Elle est utilisée tout simplement pour extraire une super classe vide.

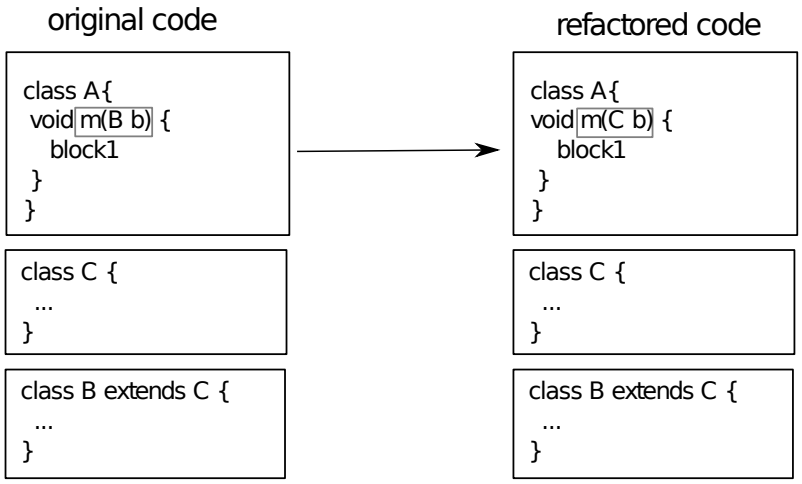
Précondition.
$(\neg\texttt{ExistsType}(s)$
$\wedge\, \texttt{ExistsClass}(a)$
$\wedge\, \texttt{ExistsClass}(b)$
$\wedge\, \texttt{ExtendsDirectly}(a, java.lang.Object)$
$\wedge\, \texttt{ExtendsDirectly}(b, java.lang.Object))$

Rétro-description.
$\texttt{IsAbstractClass}(s) \mapsto \top$
$\texttt{ExistsClass}(s) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(X, Y, [s]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(s, X, [Y]) \mapsto \bot$
$\texttt{ExistsType}(s) \mapsto \top$
$\texttt{AllSubclasses}(s, [a; b]) \mapsto \top$
$\texttt{IsUsedConstructorAsMethodParameter}(s, X, Y) \mapsto \bot$
$\texttt{IsUsedConstructorAsInitializer}(s, X, Y) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(s, X, Y) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(X, s, Y) \mapsto \bot$
$\texttt{IsUsedConstructorAsInitializer}(X, s, Y) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(X, s, Y) \mapsto \bot$

$\texttt{IsPrimitiveType}(s) \mapsto \bot$
$\texttt{IsUsedMethod}(s, X, [Y]) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(X, Y, [s]) \mapsto \bot$
$\texttt{IsPrivate}(s, X) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(X, Y, [Z], [s]) \mapsto \bot$
$\texttt{IsSubType}(a, s) \mapsto \top$
$\texttt{IsSubType}(b, s) \mapsto \top$
$\texttt{IsSubType}(X, s) \mapsto \texttt{IsSubType}(X, a)$
$\texttt{IsSubType}(X, s) \mapsto \texttt{IsSubType}(X, b)$

12 GeneraliseParameter

Rôle. GeneraliseParameter (classname s, subclasses [a,b], methodname m,paramName p,type t ,supertype st) : changer le type t du paramètre p des méthodes s : :m, a : :m and b : :m en un sur type st. Toutes les utilisations du paramètre dont le type est changé doivent être légales avec le nouveau type st (invocations de la méthode qui utilise ce paramètre ou un objet passé comme paramètre pour d'autres méthodes).



Outils de refactoring. *Change Method Signature* avec Eclipse et *Type Migration* dans IntelliJ IDEA (ou *Change Signature*).

Précondition.
$(\texttt{ExistsClass}(s)$
$\wedge\, \texttt{ExistsClass}(a)$
$\wedge\, \texttt{ExistsClass}(b)$
$\wedge\, \texttt{ExistsMethodDefinition}(s, m)$
$\wedge\, \texttt{ExistsMethodDefinition}(a, m)$
$\wedge\, \texttt{ExistsMethodDefinition}(b, m)$
$\wedge\, \texttt{IsSubType}(st, t)$
$\wedge\, \texttt{AllSubclasses}(s, [a; b])$
$\wedge\, \texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(s, m, p, t)$
$\wedge\, \texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(a, m, p, t)$
$\wedge\, \texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(b, m, p, t)$
$\wedge\, \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(s, m, p)$
$\wedge\, \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(a, m, p)$

$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded(b, m, p))

Rétro-description.
IsInverter$(s, m, t, V) \mapsto$ IsInverter(s, m, st, V)
IsInverter$(a, m, t, V) \mapsto$ IsInverter(a, m, st, V)
IsInverter$(b, m, t, V) \mapsto$ IsInverter(b, m, st, V)
ExistsMethodDefinitionWithParams$(s, m, [t]) \mapsto$ ExistsMethodDefinitionWithParams$(s, m, [st])$
ExistsMethodDefinitionWithParams$(a, m, [t]) \mapsto$ ExistsMethodDefinitionWithParams$(a, m, [st])$
ExistsMethodDefinitionWithParams$(b, m, [t]) \mapsto$ ExistsMethodDefinitionWithParams$(b, m, [st])$
ExistsMethodDefinitionWithParams$(s, m, [st]) \mapsto \bot$
ExistsMethodDefinitionWithParams$(a, m, [st]) \mapsto \bot$
ExistsMethodDefinitionWithParams$(b, m, [st]) \mapsto \bot$
IsInheritedMethodWithParams$(a, m, [t]) \mapsto \top$
IsInheritedMethodWithParams$(b, m, [t]) \mapsto \top$
IsUsedConstructorAsMethodParameter$(t, s, m) \mapsto \top$
IsUsedConstructorAsMethodParameter$(t, a, m) \mapsto \top$
IsUsedConstructorAsMethodParameter$(t, b, m) \mapsto \top$
IsUsedConstructorAsMethodParameter$(st, s, m) \mapsto \bot$
IsUsedConstructorAsMethodParameter$(st, a, m) \mapsto \bot$
IsUsedConstructorAsMethodParameter$(st, b, m) \mapsto \bot$
IsOverridden$(a, m) \mapsto$ ExistsMethodDefinition(a, m)
IsOverridden$(b, m) \mapsto$ ExistsMethodDefinition(b, m)
IsOverriding$(a, m) \mapsto$ ExistsMethodDefinition(a, m)
IsOverriding$(b, m) \mapsto$ ExistsMethodDefinition(b, m)
ExistsParameterWithName$(s, m, [t], p) \mapsto \top$
ExistsParameterWithName$(a, m, [t], p) \mapsto \top$
ExistsParameterWithName$(b, m, [t], p) \mapsto \top$
ExistsParameterWithType$(s, m, [t], t) \mapsto \top$
ExistsParameterWithType$(a, m, [t], t) \mapsto \top$
ExistsParameterWithType$(b, m, [t], t) \mapsto \top$

13 MergeDuplicateMethods

Rôle MergeDuplicateMethods (classname s, subclasses [a,b],mergedmethods [m,n],newmethod m2, invertedtype t, returntype q) : Cette opération est utilisée pour combiner deux méthodes *m* et *n* qui existent dans une hiérarchie de classes et qui sont égales de point de vue sémantique. La spécification formelle de cette opération se base sur la spécification des cinq opérations de refactoring qui la constituent.

Algorithme d'exécution Cette opération s'exécute selon l'enchainement des opérations de refactoring élémentaires suivantes :

MergeDuplicateMethods (s,[a,b],[m,n],m2,t,q) =

1. ReplaceMethodcodeDuplicatesInverter (s, m, [n], t,q))
2. PullupConcreteDelegator(s, [a,b], n ,m))
3. InlineAndDelete(s,n))
4. RenameInHierarchyNoOverloading (s, [a,b], m,[t], m2)

les opérations qui constituent cette opération sont toutes offertes par l'outil de refactoring de IntelliJ IDEA que nous utilisons et sont décrites par leurs préconditions et rétro-descriptions dans l'annexe B.

Outils de refactoring. *Rename, Replace Method duplication, Extract Method, In-line* avec Eclipse, *Rename, Replace Method Code Duplicates, Pull Up, Inline* avec IntelliJ IDEA :

Précondition.

$\texttt{ExistsMethodDefinition}(s, m)$
$\wedge\, \texttt{ExistsMethodDefinitionWithParams}(s, m, [t])$
$\wedge\, \texttt{AllSubclasses}(s, [a; b])$
$\wedge\, \neg\texttt{ExistsMethodDefinition}(s, m2)$
$\wedge\, \neg\texttt{ExistsMethodDefinition}(a, m2)$
$\wedge\, \neg\texttt{ExistsMethodDefinition}(b, m2)$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(s, m2, [t])$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(a, m2, [t])$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(b, m2, [t])$
$\wedge\, \neg\texttt{IsOverloaded}(s, m)$
$\wedge\, \neg\texttt{IsOverloaded}(a, m)$
$\wedge\, \neg\texttt{IsOverloaded}(b, m)$
$\wedge\, \neg\texttt{IsInheritedMethod}(s, m2)$
$\wedge\, \neg\texttt{IsOverriding}(s, n)$
$\wedge\, \neg\texttt{IsOverridden}(s, n)$
$\wedge\, \neg\texttt{IsRecursiveMethod}(s, n)$
$\wedge\, \neg\texttt{IsUsedMethodIn}(s, n, a)$
$\wedge\, \neg\texttt{IsUsedMethodIn}(s, n, b)$
$\wedge\, \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(b, n, this)$
$\wedge\, \texttt{ExistsClass}(s)$
$\wedge\, \texttt{IsAbstractClass}(s)$
$\wedge\, \texttt{ExistsAbstractMethod}(s, n)$
$\wedge\, \texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(a, n, this, s)$
$\wedge\, \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(a, n, this)$
$\wedge\, \neg\texttt{IsPrivate}(a, n)$
$\wedge\, \texttt{ExistsClass}(b)$
$\wedge\, \texttt{ExistsMethodDefinition}(b, m)$
$\wedge\, \texttt{ExistsMethodDefinition}(b, n)$
$\wedge\, \texttt{ExistsClass}(a)$
$\wedge\, \texttt{ExistsMethodDefinition}(a, m)$
$\wedge\, \texttt{ExistsMethodDefinition}(a, n)$
$\wedge\, \texttt{IsInverter}(b, m, t, q)$
$\wedge\, \texttt{IsInverter}(b, n, t, q)$
$\wedge\, \texttt{IsInverter}(a, m, t, q)$
$\wedge\, \texttt{IsInverter}(a, n, t, q)$

14 ReplaceMethodcodeDuplicatesInverter

Rôle ReplaceMethodcodeDuplicatesInverter (classname c, method m, copies [n,m1],invertedtype t,returntype r) :remplacer c : :[n,m1] par c : :m.

Précondition.
$(\texttt{ExistsClass}(c)$
$\wedge (\texttt{ExistsMethodDefinition}(c, m)$
$\wedge \texttt{ExistsMethodDefinition}(c, n)$
$\wedge \texttt{ExistsMethodDefinition}(c, m1))$
$\wedge (\texttt{IsInverter}(c, m, t, r)$
$\wedge \texttt{IsInverter}(c, n, t, r)$
$\wedge \texttt{IsInverter}(c, m1, t, r)))$

Rétro-description.
$\texttt{IsDelegator}(c, n, m) \mapsto \top$
$\texttt{IsDelegator}(c, m1, m) \mapsto \top$
$\texttt{ExistsMethodInvocation}(c, n, c, m) \mapsto \bot$
$\texttt{ExistsMethodInvocation}(c, m1, c, m) \mapsto \bot$
$\texttt{IsRecursiveMethod}(c, n) \mapsto \bot$
$\texttt{IsRecursiveMethod}(c, m1) \mapsto \bot$

15 SafeDeleteDelegatorOverriding

Rôle SafeDeleteDelegatorOverriding (classname c, method m, superclass s, deleguee n) : supprimer les méthodes qui sont *overridings* et qui ne sont pas utilisées.

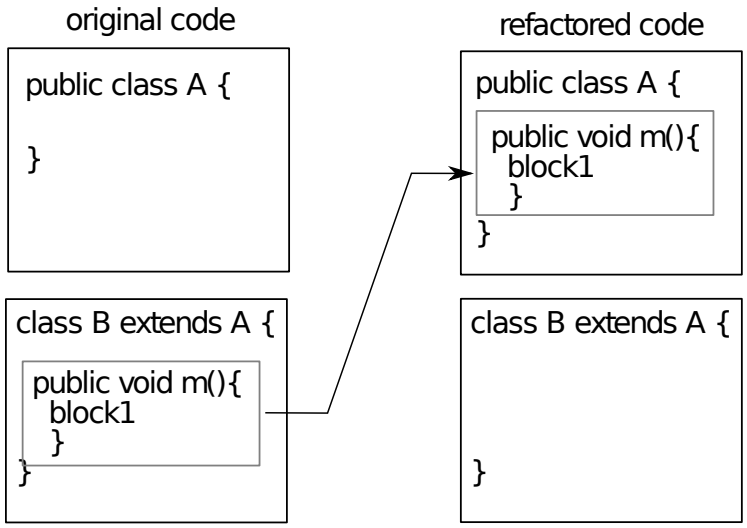
Précondition.
$(\texttt{ExistsClass}(c)$
$\wedge \texttt{ExistsClass}(s)$
$\wedge \texttt{ExistsMethodDefinition}(c, m)$
$\wedge \texttt{ExistsMethodDefinition}(s, m)$
$\wedge \texttt{IsDelegator}(c, m, n)$
$\wedge \texttt{IsDelegator}(s, m, n)$
$\wedge \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(c, m, this))$

Rétro-description.
$\texttt{ExistsMethodDefinition}(c, m) \mapsto \bot$
$\texttt{IsInheritedMethod}(c, m) \mapsto \top$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(c, m, X, Y) \mapsto \bot$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(c, m, X) \mapsto \bot$
$\texttt{BoundVariableInMethodBody}(c, m, X) \mapsto \bot$
$\texttt{ExistsParameterWithName}(c, m, [X], Y) \mapsto \bot$
$\texttt{ExistsParameterWithType}(c, m, [X], Y) \mapsto \bot$
$\texttt{ExistsMethodInvocation}(c, m, X, Y) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(c, m, [X]) \mapsto \bot$

$\texttt{IsInheritedMethodWithParams}(X, m, [Y]) \mapsto \bot$
$\texttt{IsIndirectlyRecursive}(c, m) \mapsto \bot$
$\texttt{IsVisibleMethod}(c, m, [X], Y) \mapsto \bot$
$\texttt{IsInverter}(c, m, X, Y) \mapsto \bot$
$\texttt{IsDelegator}(c, m, X) \mapsto \bot$
$\texttt{IsUsedMethod}(c, m, [X]) \mapsto \bot$
$\texttt{IsUsedMethodIn}(c, m, X) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(X, c, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsInitializer}(X, c, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(X, c, m) \mapsto \bot$
$\texttt{IsPublic}(c, m) \mapsto \bot$
$\texttt{IsProtected}(c, m) \mapsto \bot$
$\texttt{IsPrivate}(c, m) \mapsto \bot$
$\texttt{IsUsedAttributeInMethodBody}(c, X, m) \mapsto \bot$
$\texttt{IsOverridden}(c, m) \mapsto \bot$
$\texttt{IsOverloaded}(c, m) \mapsto \bot$
$\texttt{IsOverriding}(c, m) \mapsto \bot$
$\texttt{IsRecursiveMethod}(c, m) \mapsto \bot$
$\texttt{HasReturnType}(c, m, X) \mapsto \bot$
$\texttt{MethodHasParameterType}(c, m, X) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(c, m, [X], [X]) \mapsto \bot$

16 PullUpImplementation

Rôle. PullupImplementation(a,[att1,att2],m,s) : faire monter la définition de la méthode a : :m vers la classe s et la supprimer de la classe a.



Outils de refactoring. *Pull Up* avec Eclipse tool et IntelliJ IDEA.

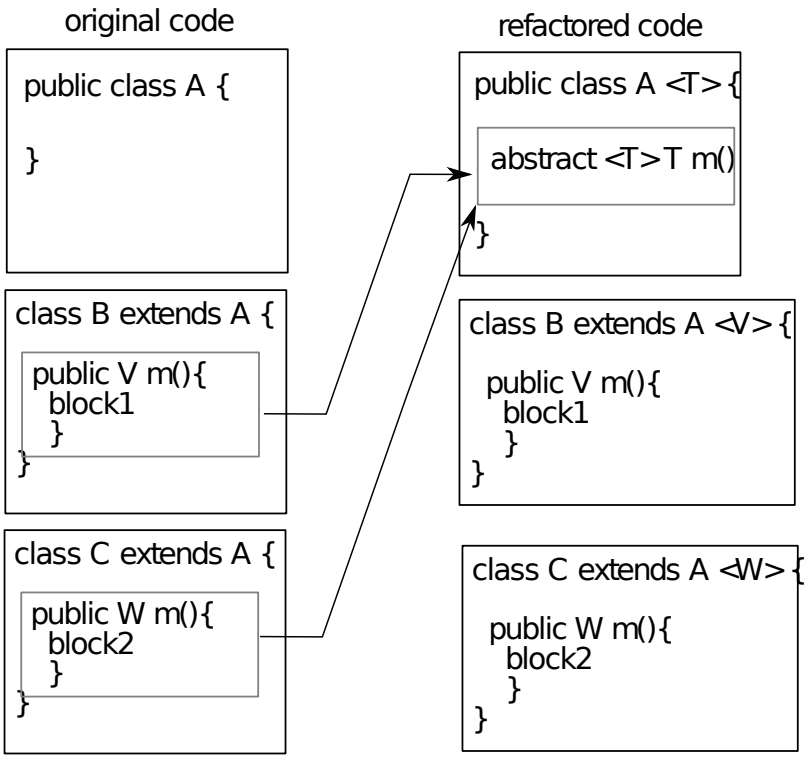
Précondition. $(\texttt{ExistsClass}(c)$
$\wedge\, \texttt{ExistsClass}(s)$
$\wedge\, \texttt{IsAbstractClass}(s)$
$\wedge\, \texttt{ExistsMethodDefinition}(c, m)$

$\wedge$ ExistsAbstractMethod(s, m)
$\wedge$ AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC$(c, m, this, s)$
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded$(c, m, this)$
$\wedge \neg$IsPrivate(c, m)
$\wedge \neg$IsUsedAttributeInMethodBody$(c, att1, m)$
$\wedge \neg$IsUsedAttributeInMethodBody$(c, att2, m))$

Rétro-description.
ExistsMethodDefinition$(c, m) \mapsto \bot$
ExistsMethodDefinition$(s, m) \mapsto \top$
ExistsAbstractMethod$(s, m) \mapsto \bot$
IsDelegator$(s, m, X) \mapsto$ IsDelegator$(c, m, X)(condition)$
ExistsMethodDefinitionWithParams$(c, m, [X]) \mapsto \bot$
IsInheritedMethodWithParams$(c, m, [X]) \mapsto \top$
IsInheritedMethod$(c, m) \mapsto \top$
IsVisibleMethod$(s, m, [X], c) \mapsto \top$
IsPrivate$(c, m) \mapsto \bot$
IsOverridden$(c, m) \mapsto \bot$
IsOverriding$(c, m) \mapsto \bot$
IsVisible$(s, m, c) \mapsto \top$
IsOverloaded$(s, m) \mapsto$ ExistsMethodDefinition(s, m)
IsUsedAttributeInMethodBody$(c, X, m) \mapsto \bot$
IsOverridden$(c, m) \mapsto \bot$
IsOverloaded$(c, m) \mapsto \bot$
IsRecursiveMethod$(c, m) \mapsto \bot$
HasReturnType$(c, m, X) \mapsto \bot$
MethodHasParameterType$(c, m, X) \mapsto \bot$
MethodIsUsedWithType$(c, m, [X], [X]) \mapsto \bot$
IsPrivate$(c, att1) \mapsto \bot$
IsPrivate$(c, att2) \mapsto \bot$

17 PullUpWithGenerics

Rôle. PullupWithGenerics (classname s, subclassname a, [att1,att2],methodname m,returntype r ,parameterType T) : faire monter la méthode a : :m vers la classe s puis créer le type paramétrique T pour la classe s (comme montré par la figure ci-dessous). Après l'application de cette opération, un polymorphisme est créé dans la hiérarchie (en Java types génériques).



Outils de refactoring. Nous proposons l'opération sous la forme d'un *plugin* pour IntelliJ IDEA (Extension de l'opération de refactoring *Pull up* : http://plugins.intellij.net/plugin/?idea_ce&id=6889).

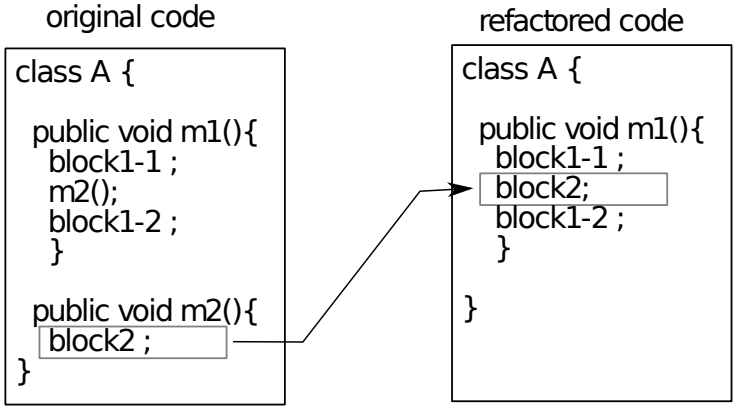
Précondition.
$(\texttt{ExistsClass}(s)$
$\wedge\ \texttt{IsAbstractClass}(s)$
$\wedge\ \texttt{ExistsClass}(a)$
$\wedge\ \texttt{IsSubType}(a, s)$
$\wedge\ \texttt{HasReturnType}(a, m, r)$
$\wedge\ \neg\texttt{ExistsAbstractMethod}(s, m)$
$\wedge\ \neg\texttt{IsPrimitiveType}(r)$
$\wedge\ \neg\texttt{IsPrivate}(a, m)$
$\wedge\ \neg\texttt{HasParameterType}(a, r)$
$\wedge\ \neg\texttt{IsPrivate}(a, att1)$
$\wedge\ \neg\texttt{IsPrivate}(a, att2))$

Rétro-description. $\texttt{HasReturnType}(s, m, T) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(s, m, [X]) \mapsto \top$
$\texttt{MethodHasParameterType}(s, m, T) \mapsto \top$
$\texttt{HasParameterType}(s, T) \mapsto \top$
$\texttt{extendsFromPrametricClass}(a, s, r) \mapsto \top$
$\texttt{IsGenericsSubtype}(a, [r], s, [T]) \mapsto \top$
$\texttt{IsPrivate}(a, m) \mapsto \bot$

18 InlineAndDelete

(*Inline Method* in [Fow99])

Rôle. InlineAndDelete (classname s,methodname m,types [t,t'],invocatormethod n, othermethods [m1,m2],otherclasses [a,b,c]) : remplacer un ou plusieurs invocations de la méthode m par sa définition et la supprimer après.



Outils de refactoring. *In-line* avec Eclipse tool et IntelliJ IDEA.

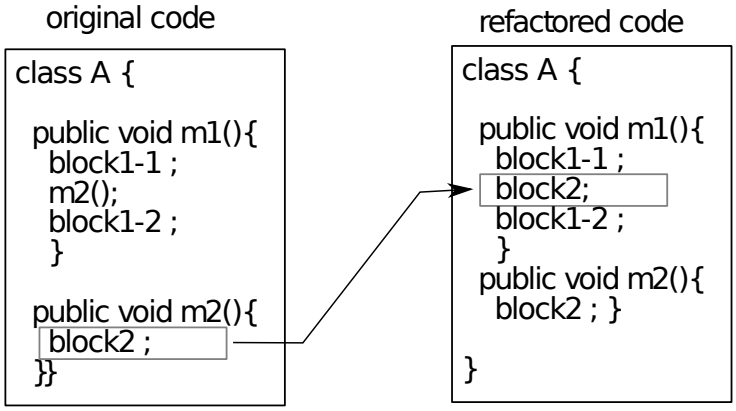
Précondition.
$(\texttt{ExistsClass}(s)$
$\wedge\, \texttt{ExistsMethodDefinition}(s, m)$
$\wedge\, \neg\texttt{IsOverriding}(s, m)$
$\wedge\, \neg\texttt{IsOverridden}(s, m)$
$\wedge\, \neg\texttt{IsRecursiveMethod}(s, m)$
$\wedge\, \neg\texttt{ExistsMethodInvocation}(s, m, s, m1)$
$\wedge\, \neg\texttt{ExistsMethodInvocation}(s, m, s, m2)$
$\wedge\, \neg\texttt{IsUsedMethodIn}(s, m, a)$
$\wedge\, \neg\texttt{IsUsedMethodIn}(s, m, b)$
$\wedge\, \neg\texttt{IsUsedMethodIn}(s, m, c))$

Rétro-description. $\texttt{ExistsMethodDefinition}(s, m) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(s, m, [t; t']) \mapsto \bot$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(s, m, X, Y) \mapsto \bot$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(s, m, X) \mapsto \bot$
$\texttt{BoundVariableInMethodBody}(s, m, X) \mapsto \bot$
$\texttt{ExistsParameterWithName}(s, m, [X], Y) \mapsto \bot$
$\texttt{ExistsParameterWithType}(s, m, [X], Y) \mapsto \bot$
$\texttt{ExistsMethodInvocation}(s, m, X, Y) \mapsto \bot$
$\texttt{ExistsMethodInvocation}(s, n, Y, Z) \mapsto$
$(\texttt{ExistsMethodInvocation}(s, m, Y, Z) \quad \vee \texttt{ExistsMethodInvocation}(s, n, Y, Z))$
$\texttt{IsUsedConstructorAsObjectReceiver}(X, s, n) \mapsto$
$(\texttt{IsUsedConstructorAsObjectReceiver}(X, s, m) \quad \vee \texttt{IsUsedConstructorAsObjectReceiver}(X, s, n))$
$\texttt{IsIndirectlyRecursive}(s, m) \mapsto \bot$
$\texttt{IsVisibleMethod}(s, m, [X], Y) \mapsto \bot$
$\texttt{IsInverter}(s, m, X, Y) \mapsto \bot$
$\texttt{IsDelegator}(s, m, X) \mapsto \bot$
$\texttt{IsUsedMethod}(s, m, [X]) \mapsto \bot$
$\texttt{IsUsedMethodIn}(s, m, X) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(X, s, m) \mapsto \bot$

$\texttt{IsUsedConstructorAsInitializer}(X, s, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(X, s, m) \mapsto \bot$
$\texttt{IsPublic}(s, m) \mapsto \bot$
$\texttt{IsProtected}(s, m) \mapsto \bot$
$\texttt{IsPrivate}(s, m) \mapsto \bot$
$\texttt{IsUsedAttributeInMethodBody}(s, X, m) \mapsto \bot$
$\texttt{IsOverridden}(s, m) \mapsto \bot$
$\texttt{IsOverloaded}(s, m) \mapsto \bot$
$\texttt{IsOverriding}(s, m) \mapsto \bot$
$\texttt{IsRecursiveMethod}(s, m) \mapsto \bot$
$\texttt{HasReturnType}(s, m, X) \mapsto \bot$
$\texttt{HasParameterType}(s, m) \mapsto \bot$
$\texttt{MethodHasParameterType}(s, m, X) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(s, m, [X], [X]) \mapsto \bot$

19 InlineMethodInvocations

Rôle. InlineMethodInvocations(classname c,inlinedmethod m, classofinlinedmethod a, modifiedmethod n) : faire un *inline* pour l'invocation de la méthode c : :m dans la portée de la méthode a : :n.



Outils de refactoring. Inline avec Eclipse et IntelliJ IDEA : sélectionner l'invocation à y faire un *inline*.

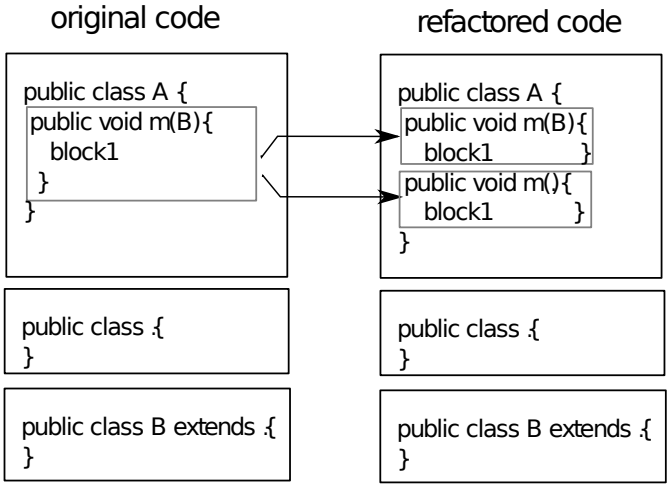
Précondition.
$(\texttt{ExistsClass}(c)$
$\wedge\ \texttt{IsIndirectlyRecursive}(c, m)$
$\wedge\ \texttt{IsRecursiveMethod}(c, n)$
$\wedge\ \neg\texttt{IsRecursiveMethod}(c, m)$
$\wedge\ \texttt{ExistsMethodInvocation}(c, m, a, n)$
$\wedge\ \texttt{ExistsMethodDefinition}(c, m)$
$\wedge\ \texttt{ExistsMethodDefinitionWithParams}(c, m, [t; t'])$
$\wedge\ \texttt{ExistsMethodDefinition}(a, n)$
$\wedge\ \texttt{ExistsMethodDefinitionWithParams}(a, n, [t1; t1']))$

Rétro-description. $\texttt{ExistsMethodDefinitionWithParams}(c, m, [t; t']) \mapsto \top$
$\texttt{ExistsMethodDefinition}(c, m) \mapsto \top$

$\texttt{ExistsMethodInvocation}(c, m, a, n) \mapsto \bot$
$\texttt{IsRecursiveMethod}(c, m) \mapsto \texttt{ExistsMethodInvocation}(a, n, c, m)$
$\texttt{IsIndirectlyRecursive}(c, m) \mapsto$
$(\texttt{ExistsMethodInvocation}(a, n, C, X)$
$\wedge \texttt{ExistsMethodInvocation}(C, X, c, m))(condition)$
$\texttt{IsUsedMethodIn}(a, n, m) \mapsto \bot$

20 AddSpecializedMethodInHierarchy (Composée)

Rôle. AddSpecializedMethodInHierarchy(class s, subclasses [a,b], methodname m, callermethods [n,o], inkovekmethods [p,q], paramtype t, paramname pn , subtypesOfparamtype [t1,t2],newtype t') : obtenir la méthode s : :m(t' pn) instead à partir de la méthode s : :m(t pn). La nouvelle duplication sera existante dans toute la hiérarchie.



Algorithm of the operation L'opération AddSpecializedMethodInHierarchy est basée sur trois étapes :

AddSpecializedMethodInHierarchy(class s, subclasses [a,b], methodname m, callermethods [n,o], inkovekmethods [p,q], paramtype t, paramname pn , subtypesOfparamtype [t1,t2],newtype t') =

1. DuplicateMethodInHierarchy s [a,b] m [p,q] [n,o] temporaryName [t]
2. SpecialiseParameter s [a,b] temporaryName t pn [t1,t2] t' ;
3. RenameDelegatorWithOverloading (s, [a,b], temporaryName,t', pn,t,m)

Outils de refactoring. avec IntelliJ IDEA :

1. Appliquer DuplicateMethodInHierarchy(c, m, temp-name) (see 21 ci-dessous).
2. Appliquer *Change Signature* sur la méthode temp-name dans la classe s, pour changer le type t en t' (ce changement est propagé dans les sous classes).

   Noter que la sémantique n'est pas garantie par cette opération en général, mais ici nous introduisons une nouvelle méthode ce qui ne viole pas la sémantique.
3. Renommer la méthode temp-name en m dans la classe s avec l'opération *Rename*.

21 DuplicateMethodInHierarchy

Rôle. DuplicateMethodInHierarchy(class s, subclasses [a,b], methodname m, callermethods [m1,m2], inkovekmethods [m3,m4],newname n ,paramType [t]) : créer une duplication de la méthode s : :m avec le nom n. Toutes les méthodes qui redéfinissent la méthode s : :m vont avoir aussi des duplications.

Outils de refactoring. Avec IntelliJ IDEA :

1. Pour chaque définition de la méthode m dans la hiérarchie, dupliquer cette méthode par l'application de l'opération *Extract Method* sur son corps (donner un nouveau nom et préciser la nouvelle visiblité), puis faire un *inline* pour l'invocation de n qui est le nom de la méthode extraite de m.

2. Utiliser l'opération *Pull Members Up* pour faire apparaitre la nouvelle méthode dans les classes où la méthode initiale est déclarée abstraite (la nouvelle méthode doit être aussi déclarée abstraite)

Précondition.
$(\texttt{ExistsClass}(s)$
$\wedge\, \texttt{ExistsMethodDefinitionWithParams}(s, m, [t; t'])$
$\wedge\, \texttt{ExistsMethodDefinition}(s, m)$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(s, n, [t; t'])$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(a, n, [t; t'])$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(b, n, [t; t'])$
$\wedge\, \neg\texttt{IsInheritedMethodWithParams}(s, n, [t; t'])$
$\wedge\, \texttt{AllSubclasses}(s, [a; b]))$

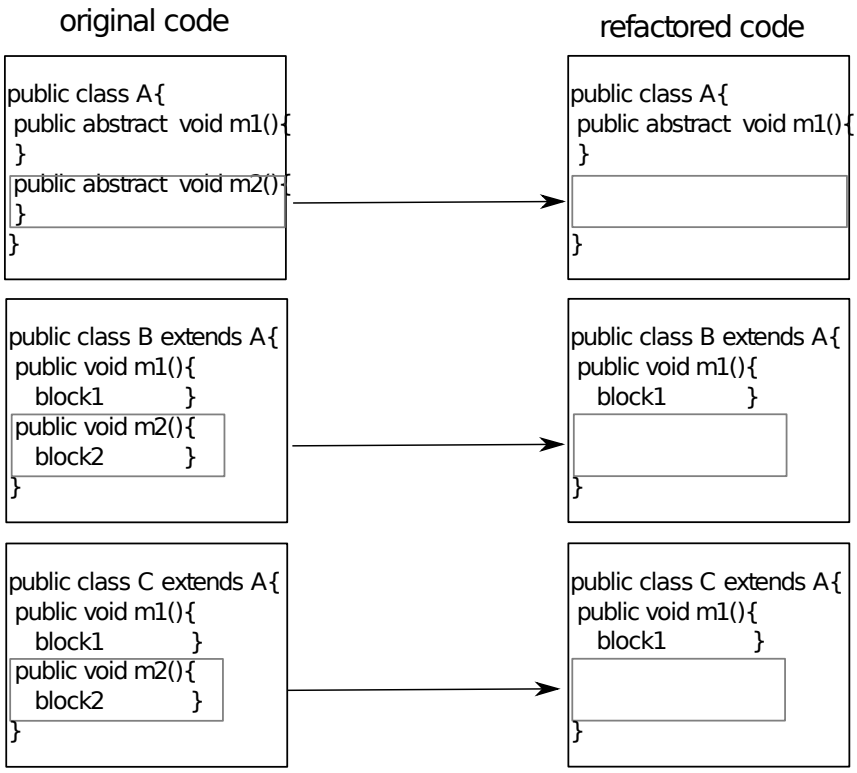
Rétro-description.
$\texttt{ExistsMethodDefinition}(s, n) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(s, n, [t; t']) \mapsto \top$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(s, n, V) \mapsto \top(condition)$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(s, n, V, V1)$
$\quad \mapsto \texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(s, m, V, V1)$
$\texttt{BoundVariableInMethodBody}(s, n, V) \mapsto \texttt{BoundVariableInMethodBody}(s, m, V)$
$\texttt{IsPublic}(s, n) \mapsto \texttt{IsPublic}(s, m)$
$\texttt{ExistsParameterWithName}(s, n, [t; t'], V) \mapsto \texttt{ExistsParameterWithName}(s, m, [t; t'], V)$
$\texttt{ExistsParameterWithType}(s, n, [t; t'], T) \mapsto \texttt{ExistsParameterWithType}(s, m, [t; t'], T)$
$\texttt{IsIndirectlyRecursive}(s, n) \mapsto \texttt{IsIndirectlyRecursive}(s, m)$
$\texttt{IsRecursiveMethod}(s, n) \mapsto \texttt{IsRecursiveMethod}(s, m)$
$\texttt{IsInverter}(s, n, T, V) \mapsto \texttt{IsInverter}(s, m, T, V)$
$\texttt{IsUsedAttributeInMethodBody}(s, V, n)$
$\quad \mapsto \texttt{IsUsedAttributeInMethodBody}(s, V, m)$
$\texttt{MethodHasParameterType}(s, n, V)$
$\quad \mapsto \texttt{MethodHasParameterType}(s, m, V)$
$\texttt{ExistsMethodDefinitionWithParams}(a, n, [t; t'])$
$\quad \mapsto \texttt{ExistsMethodDefinitionWithParams}(a, m, [t; t'])$
$\texttt{ExistsMethodDefinitionWithParams}(b, n, [t; t'])$
$\quad \mapsto \texttt{ExistsMethodDefinitionWithParams}(b, m, [t; t'])$
$\texttt{IsDelegator}(s, n, m3) \mapsto \top$
$\texttt{IsDelegator}(s, n, m4) \mapsto \top$
$\texttt{IsDelegator}(a, n, m3) \mapsto \top$

$\texttt{IsDelegator}(a, n, m4) \mapsto \top$
$\texttt{IsDelegator}(b, n, m3) \mapsto \top$
$\texttt{IsDelegator}(b, n, m4) \mapsto \top$
$\texttt{ExistsMethodDefinition}(s, n) \mapsto \top$
$\texttt{ExistsMethodDefinition}(a, n) \mapsto \top$
$\texttt{ExistsMethodDefinition}(b, n) \mapsto \top$
$\texttt{MethodIsUsedWithType}(s, n, [t; t'], [t; t']) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(a, n, [t; t'], [t; t']) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(b, n, [t; t'], [t; t']) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(s, n, [t; t'], [T]) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(a, n, [t; t'], [T]) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(b, n, [t; t'], [T]) \mapsto \bot$
$\texttt{ExistsMethodInvocation}(s, m1, V, n) \mapsto \top$
$\texttt{ExistsMethodInvocation}(s, m2, V, n) \mapsto \top$
$\texttt{ExistsMethodInvocation}(a, m1, V, n) \mapsto \top$
$\texttt{ExistsMethodInvocation}(a, m2, V, n) \mapsto \top$
$\texttt{ExistsMethodInvocation}(b, m1, V, n) \mapsto \top$
$\texttt{ExistsMethodInvocation}(b, m2, V, n) \mapsto \top$
$\texttt{IsInheritedMethodWithParams}(a, n, [t; t'])$
$\quad \mapsto \neg\texttt{ExistsMethodDefinitionWithParams}(a, m, [t; t'])$
$\texttt{IsInheritedMethodWithParams}(b, n, [t; t'])$
$\quad \mapsto \neg\texttt{ExistsMethodDefinitionWithParams}(b, m, [t; t'])$
$\texttt{IsInheritedMethod}(a, n) \mapsto \neg\texttt{ExistsMethodDefinition}(a, m)$
$\texttt{IsInheritedMethod}(b, n) \mapsto \neg\texttt{ExistsMethodDefinition}(b, m)$
$\texttt{IsOverriding}(a, n) \mapsto \neg\texttt{ExistsMethodDefinition}(a, m)$
$\texttt{IsOverriding}(b, n) \mapsto \neg\texttt{ExistsMethodDefinition}(b, m)$
$\texttt{IsOverridden}(a, n) \mapsto \neg\texttt{ExistsMethodDefinition}(a, m)$
$\texttt{IsOverridden}(b, n) \mapsto \neg\texttt{ExistsMethodDefinition}(b, m)$

22 DeleteMethodInHierarchy

(*Delete Method* dans Fowler [Fow99] et [Koc02])

Rôle. DeleteMethodInHierarchy (classname s, subclasses [a,b], method m, invokedmethodsInm [m1,m2], paramType t) : supprimer la méthode m de toute la hiérarchie de classe s,a et b.



Outils de refactoring. *Safe Delete* avec IntelliJ IDEA et *Delete* in Eclipse.

Précondition.
$(\texttt{ExistsClass}(s)$
$\wedge\ \texttt{ExistsMethodDefinitionWithParams}(s, m, [t])$
$\wedge\ \neg\texttt{MethodIsUsedWithType}(s, m, [t], [t])$
$\wedge\ \neg\texttt{MethodIsUsedWithType}(a, m, [t], [t])$
$\wedge\ \neg\texttt{MethodIsUsedWithType}(b, m, [t], [t])$
$\wedge\ \texttt{AllSubclasses}(s, [a; b]))$

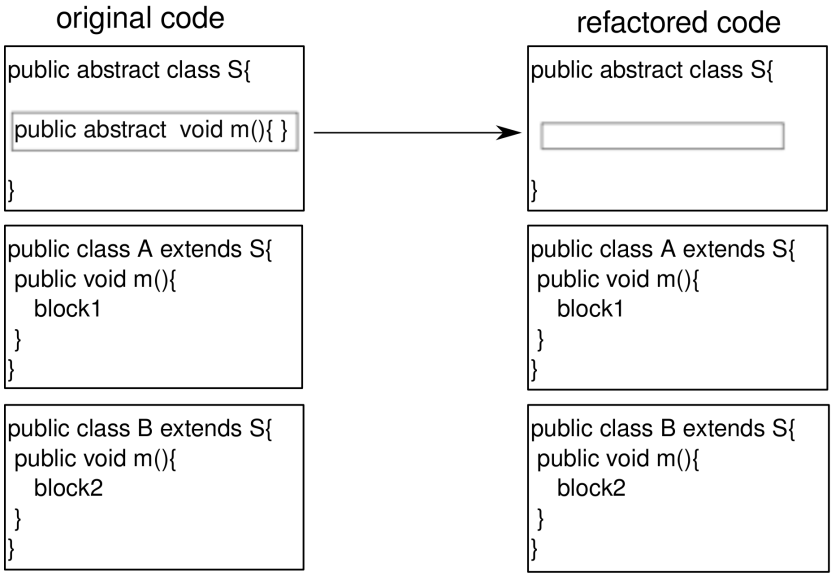
Rétro-description.
$\texttt{ExistsParameterWithType}(s, m, [t], t) \mapsto \bot$
$\texttt{ExistsParameterWithType}(a, m, [t], t) \mapsto \bot$
$\texttt{ExistsParameterWithType}(b, m, [t], t) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(s, m, [t]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(a, m, [t]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(b, m, [t]) \mapsto \bot$
$\texttt{ExistsMethodDefinition}(s, m) \mapsto \bot$
$\texttt{ExistsMethodDefinition}(a, m) \mapsto \bot$
$\texttt{ExistsMethodDefinition}(b, m) \mapsto \bot$
$\texttt{IsUsedMethod}(s, m1, [V1]) \mapsto \bot$
$\texttt{IsUsedMethod}(s, m2, [V1]) \mapsto \bot$
$\texttt{IsUsedMethod}(a, m1, [V1]) \mapsto \bot$
$\texttt{IsUsedMethod}(a, m2, [V1]) \mapsto \bot$
$\texttt{IsUsedMethod}(b, m1, [V1]) \mapsto \bot$
$\texttt{IsUsedMethod}(b, m2, [V1]) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(V1, s, m) \mapsto \bot$

$\texttt{IsUsedConstructorAsMethodParameter}(V1, a, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(V1, b, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(t, s, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(t, a, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(t, b, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(t, s, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(t, a, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(t, b, m) \mapsto \bot$
$\texttt{IsInheritedMethod}(a, m) \mapsto \bot$
$\texttt{IsInheritedMethod}(b, m) \mapsto \bot$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(s, m, V1, V2) \mapsto \bot$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(a, m, V1, V2) \mapsto \bot$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(b, m, V1, V2) \mapsto \bot$
$\texttt{ExistsAbstractMethod}(s, m) \mapsto \bot$
$\texttt{ExistsAbstractMethod}(a, m) \mapsto \bot$
$\texttt{ExistsAbstractMethod}(b, m) \mapsto \bot$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(s, m, V1) \mapsto \bot$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(a, m, V1) \mapsto \bot$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(b, m, V1) \mapsto \bot$
$\texttt{BoundVariableInMethodBody}(s, m, V1) \mapsto \bot$
$\texttt{BoundVariableInMethodBody}(a, m, V1) \mapsto \bot$
$\texttt{BoundVariableInMethodBody}(b, m, V1) \mapsto \bot$
$\texttt{ExistsParameterWithName}(s, m, [t], V1) \mapsto \bot$
$\texttt{ExistsParameterWithName}(a, m, [t], V1) \mapsto \bot$
$\texttt{ExistsParameterWithName}(b, m, [t], V1) \mapsto \bot$
$\texttt{ExistsMethodInvocation}(s, m, V1, V2) \mapsto \bot$
$\texttt{ExistsMethodInvocation}(a, m, V1, V2) \mapsto \bot$
$\texttt{ExistsMethodInvocation}(b, m, V1, V2) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(a, m, [t]) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(b, m, [t]) \mapsto \bot$
$\texttt{IsIndirectlyRecursive}(s, m) \mapsto \bot$
$\texttt{IsIndirectlyRecursive}(a, m) \mapsto \bot$
$\texttt{IsIndirectlyRecursive}(b, m) \mapsto \bot$
$\texttt{IsVisibleMethod}(s, m, [t], V1) \mapsto \bot$
$\texttt{IsVisibleMethod}(a, m, [t], V1) \mapsto \bot$
$\texttt{IsVisibleMethod}(b, m, [t], V1) \mapsto \bot$
$\texttt{IsInverter}(s, m, V1, V2) \mapsto \bot$
$\texttt{IsInverter}(a, m, V1, V2) \mapsto \bot$
$\texttt{IsInverter}(b, m, V1, V2) \mapsto \bot$
$\texttt{IsDelegator}(s, V1, m) \mapsto \bot$
$\texttt{IsDelegator}(a, V1, m) \mapsto \bot$
$\texttt{IsDelegator}(b, V1, m) \mapsto \bot$
$\texttt{IsUsedMethodIn}(s, m, V1) \mapsto \bot$
$\texttt{IsUsedMethodIn}(a, m, V1) \mapsto \bot$
$\texttt{IsUsedMethodIn}(b, m, V1) \mapsto \bot$
$\texttt{IsUsedConstructorAsInitializer}(V1, s, m) \mapsto \bot$

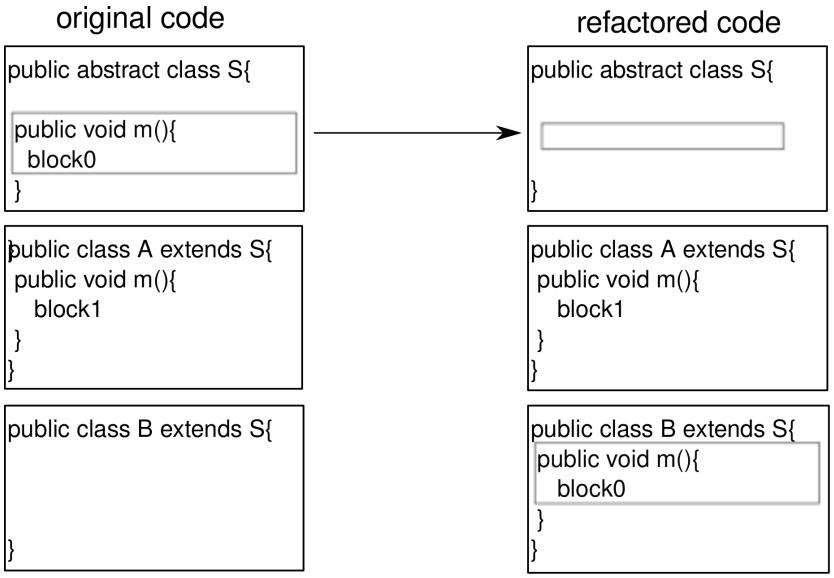
$\texttt{IsUsedConstructorAsInitializer}(V1, a, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsInitializer}(V1, b, m) \mapsto \bot$
$\texttt{IsUsedAttributeInMethodBody}(s, V1, m) \mapsto \bot$
$\texttt{IsUsedAttributeInMethodBody}(a, V1, m) \mapsto \bot$
$\texttt{IsUsedAttributeInMethodBody}(b, V1, m) \mapsto \bot$
$\texttt{IsOverridden}(a, m) \mapsto \bot$
$\texttt{IsOverridden}(b, m) \mapsto \bot$
$\texttt{IsOverloaded}(s, m) \mapsto \bot$
$\texttt{IsOverloaded}(a, m) \mapsto \bot$
$\texttt{IsOverloaded}(b, m) \mapsto \bot$
$\texttt{IsOverriding}(a, m) \mapsto \bot$
$\texttt{IsOverriding}(b, m) \mapsto \bot$
$\texttt{IsRecursiveMethod}(s, m) \mapsto \bot$
$\texttt{IsRecursiveMethod}(a, m) \mapsto \bot$
$\texttt{IsRecursiveMethod}(b, m) \mapsto \bot$
$\texttt{HasReturnType}(s, m, V1) \mapsto \bot$
$\texttt{HasReturnType}(a, m, V1) \mapsto \bot$
$\texttt{HasReturnType}(b, m, V1) \mapsto \bot$
$\texttt{MethodHasParameterType}(s, m, V1) \mapsto \bot$
$\texttt{MethodHasParameterType}(a, m, V1) \mapsto \bot$
$\texttt{MethodHasParameterType}(b, m, V1) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(s, m, [t], [t]) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(a, m, [t], [t]) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(b, m, [t], [t]) \mapsto \bot$

23 PushDownAll

Rôle. PushDownAll (classname s, attributes [att1,att2], subclasses [a,b], method m,paramType [t]) : faire descendre la méthode s : :m vers les sous classes de s et les supprimer de s (Dans l'opération *Push Down Method* de Fowler [Fow99], les méthodes à faire descendre peuvent ne pas être poussées vers toutes les sous classes).



Variation for non-abstract methods :



Outils de refactoring. *Push Down* ou *Push member Down* avec Eclipse et IntelliJ IDEA.

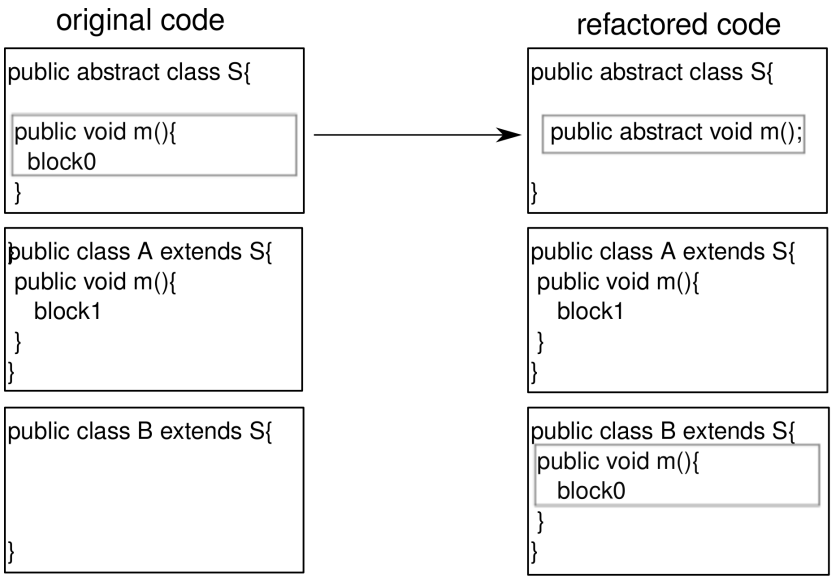
Précondition.
$(\texttt{ExistsClass}(s)$
$\wedge\ \texttt{IsAbstractClass}(s)$
$\wedge\ \texttt{ExistsMethodDefinitionWithParams}(s, m, [t])$
$\wedge\ \neg\texttt{IsUsedMethod}(s, m, [t])$
$\wedge\ \texttt{AllSubclasses}(s, [a; b])$
$\wedge\ \neg\texttt{IsPrivate}(s, m)$
$\wedge\ \texttt{ExistsMethodDefinitionWithParams}(s, m, [t])$
$\wedge\ \texttt{ExistsMethodDefinitionWithParams}(s, m, [t])$
$\wedge\ \neg\texttt{IsPrivate}(s, att1)$
$\wedge\ \neg\texttt{IsPrivate}(s, att2))$

Rétro-description. $\texttt{ExistsMethodDefinitionWithParams}(s, m, [t]) \mapsto \bot$
$\texttt{IsUsedMethodIn}(s, m, C) \mapsto \bot$
$\texttt{ExistsMethodDefinition}(s, m) \mapsto \bot$
$\texttt{ExistsAbstractMethod}(s, m) \mapsto \bot$
$\texttt{IsDelegator}(s, m, V1) \mapsto \bot$
$\texttt{HasReturnType}(s, m, V1) \mapsto \bot$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(s, m, V1, V2) \mapsto \bot$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(s, m, V1) \mapsto \bot$
$\texttt{BoundVariableInMethodBody}(s, m, V1) \mapsto \bot$
$\texttt{ExistsParameterWithName}(s, m, [t], V1) \mapsto \bot$
$\texttt{ExistsParameterWithType}(s, m, [t], V1) \mapsto \bot$
$\texttt{ExistsMethodInvocation}(s, m, V1, V2) \mapsto \bot$
$\texttt{IsPublic}(s, m) \mapsto \bot$
$\texttt{IsProtected}(s, m) \mapsto \bot$
$\texttt{IsPrivate}(s, m) \mapsto \bot$
$\texttt{IsOverloaded}(s, m) \mapsto \bot$

$\texttt{IsUsedAttributeInMethodBody}(s, V1, m) \mapsto \bot$
$\texttt{IsRecursiveMethod}(s, m) \mapsto \bot$
$\texttt{IsIndirectlyRecursive}(s, m) \mapsto \bot$
$\texttt{HasReturnType}(s, m, V1) \mapsto \bot$
$\texttt{MethodHasParameterType}(s, m, V1) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(s, m, [t], [t]) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(V1, s, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(V1, s, m) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(a, m, [t]) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(b, m, [t]) \mapsto \top$
$\texttt{ExistsMethodDefinition}(a, m) \mapsto \top$
$\texttt{ExistsMethodDefinition}(b, m) \mapsto \top$
$\texttt{IsOverriding}(a, m) \mapsto \bot$
$\texttt{IsOverriding}(b, m) \mapsto \bot$
$\texttt{IsOverridden}(a, m) \mapsto \bot$
$\texttt{IsOverridden}(b, m) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(a, m, [t]) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(b, m, [t]) \mapsto \bot$
$\texttt{IsVisibleMethod}(s, m, [t], a) \mapsto \bot$
$\texttt{IsVisibleMethod}(s, m, [t], b) \mapsto \bot$
$\texttt{IsVisible}(s, m, a) \mapsto \bot$
$\texttt{IsVisible}(s, m, b) \mapsto \bot$
$\texttt{IsInheritedMethod}(a, m) \mapsto \bot$
$\texttt{IsInheritedMethod}(b, m) \mapsto \bot$
$\texttt{IsPrivate}(s, att1) \mapsto \bot$
$\texttt{IsPrivate}(s, att2) \mapsto \bot$

24 PushDownImplementation

Rôle. PushDownImplementation (classname s, attributes [att1,att2], subclasses [a,b], method m,paramType [t,t']) : même que *PushDownAll* mais garder la méthode en question abstraite dans la classe s.



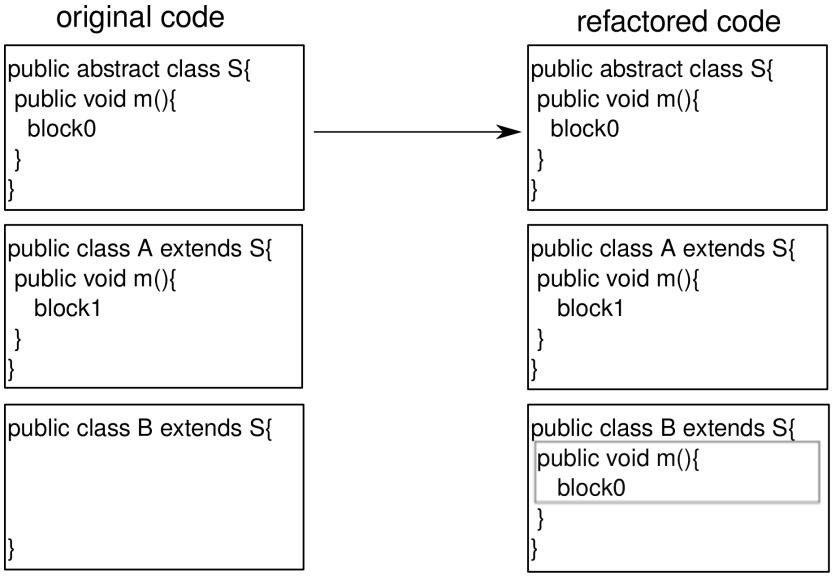
Précondition.
$(\texttt{ExistsClass}(s)$
$\wedge\, \texttt{ExistsMethodDefinition}(s, m)$
$\wedge\, \texttt{ExistsMethodDefinitionWithParams}(s, m, [t; t'])$
$\wedge\, \neg\texttt{ExistsAbstractMethod}(s, m)$
$\wedge\, \texttt{AllSubclasses}(s, [a; b])$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(a, m, [t; t'])$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(b, m, [t; t'])$
$\wedge\, \neg\texttt{IsPrivate}(s, att1)$
$\wedge\, \neg\texttt{IsPrivate}(s, att2))$

Rétro-description. $\texttt{ExistsAbstractMethod}(s, m) \mapsto \top$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(s, m, V1, V2) \mapsto \top$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(s, m, V1) \mapsto \top$
$\texttt{BoundVariableInMethodBody}(s, m, V1) \mapsto \bot$
$\texttt{ExistsMethodInvocation}(s, m, V1, V2) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(s, m, [t; t']) \mapsto \bot$
$\texttt{IsIndirectlyRecursive}(s, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsInitializer}(V1, s, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(V1, s, m) \mapsto \bot$
$\texttt{IsPrivate}(s, m) \mapsto \bot$
$\texttt{IsUsedAttributeInMethodBody}(s, V1, m) \mapsto \bot$
$\texttt{IsOverridden}(s, m) \mapsto \bot$
$\texttt{IsOverriding}(s, m) \mapsto \bot$
$\texttt{IsRecursiveMethod}(s, m) \mapsto \bot$
$\texttt{MethodHasParameterType}(s, m, V1) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(s, m, [t; t'], [t; t']) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(a, m, [t; t']) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(b, m, [t; t']) \mapsto \top$
$\texttt{ExistsMethodDefinition}(a, m) \mapsto \top$
$\texttt{ExistsMethodDefinition}(b, m) \mapsto \top$

$\texttt{IsPrivate}(s, att1) \mapsto \bot$
$\texttt{IsPrivate}(s, att2) \mapsto \bot$

25 PushDownCopy

PushDownCopy(classname c, superclass s ,notredefinedmethods [m1,m2])
Dupliquer la méthode c : :m vers les sous classes de c.



Outils de refactoring. Cette opération n'est pas encore implémentée.

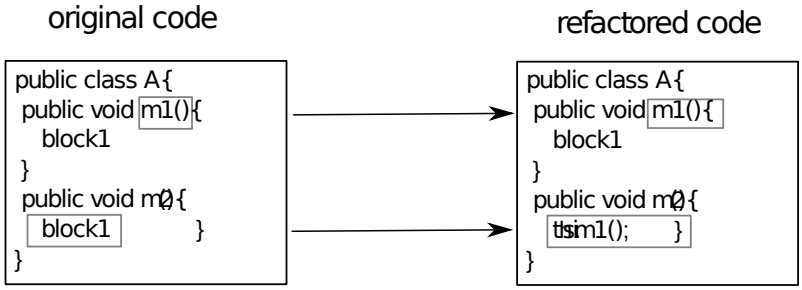
Précondition.
$\texttt{ExistsType}(c)$
$\wedge\, \texttt{ExistsClass}(c)$
$\wedge\, \texttt{IsSubType}(c, s)$
$\wedge\, \neg\texttt{ExistsMethodDefinition}(c, m1)$
$\wedge\, \neg\texttt{ExistsMethodDefinition}(c, m2)$
$\wedge\, \texttt{ExistsMethodDefinition}(s, m1)$
$\wedge\, \texttt{ExistsMethodDefinition}(s, m2)$
$\wedge\, \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(s, m1, this)$
$\wedge\, \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(s, m2, this)$

Rétro-description.
$\texttt{ExistsMethodDefinition}(c, m1) \mapsto \top$
$\texttt{ExistsMethodDefinition}(c, m2) \mapsto \top$
$\texttt{IsOverriding}(c, m1) \mapsto \top$
$\texttt{IsOverriding}(c, m2) \mapsto \top$
$\texttt{IsOverridden}(c, m1) \mapsto \top$
$\texttt{IsOverridden}(c, m2) \mapsto \top$
$\texttt{BoundVariableInMethodBody}(c, m1, V) \mapsto \texttt{BoundVariableInMethodBody}(s, m1, V)$
$\texttt{BoundVariableInMethodBody}(c, m2, V) \mapsto \texttt{BoundVariableInMethodBody}(s, m2, V)$
$\texttt{IsDuplicate}(c, m1, s, m1) \mapsto \top$

$\texttt{IsDuplicate}(c, m2, s, m2) \mapsto \top$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(c, m1, this) \mapsto \texttt{AllInvokedMethodsWit}$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(c, m2, this) \mapsto \texttt{AllInvokedMethodsWit}$

26 ReplaceMethodDuplication

Rôle. ReplaceMethodDuplication (classname s, subclasses [a,b], method m,copy n, paramType [t]) :remplacer les occurrences du corps de la méthode s : :m par m partout dans le programme.



Outils de refactoring. *Replace Method Duplication* avec IntelliJ IDEA.

Précondition.
$(\texttt{ExistsClass}(s)$
$\wedge\, \texttt{ExistsMethodDefinition}(s, m)$
$\wedge\, \texttt{ExistsMethodDefinition}(s, n)$
$\wedge\, \texttt{IsDelegator}(s, n, m)$
$\wedge\, \texttt{AllSubclasses}(s, [a; b]))$

Rétro-description.
$\texttt{IsUsedMethod}(s, n, [t]) \mapsto \bot$
$\texttt{IsDelegator}(s, n, m) \mapsto \top$
$\texttt{ExistsMethodInvocation}(s, n, s, m) \mapsto \bot$
$\texttt{IsRecursiveMethod}(s, n) \mapsto \bot$
$\texttt{IsRecursiveMethod}(s, n) \mapsto \bot$
$\texttt{IsRecursiveMethod}(a, n) \mapsto \bot$
$\texttt{IsRecursiveMethod}(b, n) \mapsto \bot$
$\texttt{ExistsMethodInvocation}(a, m, a, n) \mapsto \texttt{ExistsMethodInvocation}(a, n, a, m)$
$\texttt{ExistsMethodInvocation}(b, m, b, n) \mapsto \texttt{ExistsMethodInvocation}(b, n, b, m)$

27 DeleteClass

Rôle. DeleteClass (classname a, classnamesuperclass s ,allclasses [s,a,b], classnamemethods [m,m1],othermethods [m2,n]) : supprimer la classe a après vérifier qu'elle n'es plus utilisée.

Outils de refactoring. *Safe Delete* avec IntelliJ IDEA, *Delete* avec Eclipse.

Précondition.
$(\texttt{ExistsClass}(a)$
$\wedge\, \texttt{ExistsType}(a)$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(s, m2, [a])$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(s, n, [a])$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(b, m2, [a])$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(b, n, [a])$
$\wedge\, \neg\texttt{IsUsedMethodIn}(a, m, s)$
$\wedge\, \neg\texttt{IsUsedMethodIn}(a, m1, s)$
$\wedge\, \neg\texttt{IsUsedMethodIn}(a, m, b)$
$\wedge\, \neg\texttt{IsUsedMethodIn}(a, m1, b)$
$\wedge\, \neg\texttt{IsUsedConstructorAsMethodParameter}(a, s, m2)$
$\wedge\, \neg\texttt{IsUsedConstructorAsMethodParameter}(a, s, n)$
$\wedge\, \neg\texttt{IsUsedConstructorAsMethodParameter}(a, b, m2)$
$\wedge\, \neg\texttt{IsUsedConstructorAsMethodParameter}(a, b, n)$
$\wedge\, \neg\texttt{IsUsedConstructorAsInitializer}(a, s, m2)$
$\wedge\, \neg\texttt{IsUsedConstructorAsInitializer}(a, s, n)$
$\wedge\, \neg\texttt{IsUsedConstructorAsInitializer}(a, b, m2)$
$\wedge\, \neg\texttt{IsUsedConstructorAsInitializer}(a, b, n)$
$\wedge\, \neg\texttt{IsUsedConstructorAsObjectReceiver}(a, s, m2)$
$\wedge\, \neg\texttt{IsUsedConstructorAsObjectReceiver}(a, s, n)$
$\wedge\, \neg\texttt{IsUsedConstructorAsObjectReceiver}(a, b, m2)$
$\wedge\, \neg\texttt{IsUsedConstructorAsObjectReceiver}(a, b, n)$
$\wedge\, \neg\texttt{IsSubType}(s, a)$
$\wedge\, \neg\texttt{IsSubType}(b, a))$

Rétro-description.
$\texttt{ExistsType}(a) \mapsto \bot$
$\texttt{ExistsClass}(a) \mapsto \bot$
$\texttt{IsSubType}(a, s) \mapsto \bot$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(a, V1, V2, V3) \mapsto \bot$
$\texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(a, V1, V2) \mapsto \bot$
$\texttt{BoundVariableInMethodBody}(a, V1, V2) \mapsto \bot$
$\texttt{ExistsParameterWithName}(a, V1, [V2], V3) \mapsto \bot$
$\texttt{ExistsParameterWithType}(a, V1, [V2], V3) \mapsto \bot$
$\texttt{ExistsField}(a, V1) \mapsto \bot$
$\texttt{ExistsMethodInvocation}(a, V1, V2, V3) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(a, V1, [V2]) \mapsto \bot$
$\texttt{ExtendsDirectly}(a, s) \mapsto \bot$
$\texttt{ExtendsDirectly}(V1, a) \mapsto \bot$
$\texttt{ExistsAbstractMethod}(a, V1) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(a, V1, [V2]) \mapsto \bot$
$\texttt{IsIndirectlyRecursive}(a, V1) \mapsto \bot$
$\texttt{IsVisibleMethod}(a, V1, [V2], V3) \mapsto \bot$
$\texttt{IsInverter}(a, V1, V2, V3) \mapsto \bot$
$\texttt{IsDelegator}(a, V1, V2) \mapsto \bot$
$\texttt{IsAbstractClass}(a) \mapsto \bot$
$\texttt{IsUsedMethod}(a, V1, [V2]) \mapsto \bot$

$\texttt{IsUsedMethodIn}(a, V1, V2) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(V1, a, V2) \mapsto \bot$
$\texttt{IsUsedConstructorAsInitializer}(a, V1, V2) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(a, V1, V2) \mapsto \bot$
$\texttt{IsUsedConstructorAsInitializer}(V1, a, V2) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(V1, a, V2) \mapsto \bot$
$\texttt{IsPrimitiveType}(a) \mapsto \bot$
$\texttt{IsPublic}(a, V1) \mapsto \bot$
$\texttt{IsProtected}(a, V1) \mapsto \bot$
$\texttt{IsPrivate}(a, V1) \mapsto \bot$
$\texttt{IsUsedAttributeInMethodBody}(a, V1, V2) \mapsto \bot$
$\texttt{IsGenericsSubtype}(a, [V1], s, [V2]) \mapsto \bot$
$\texttt{IsGenericsSubtype}(V1, [V2], a, [V3]) \mapsto \bot$
$\texttt{IsGenericsSubtype}(V1, [a], V2, [V3]) \mapsto \bot$
$\texttt{IsInheritedField}(a, V1) \mapsto \bot$
$\texttt{IsOverridden}(a, V1) \mapsto \bot$
$\texttt{IsOverloaded}(a, V1) \mapsto \bot$
$\texttt{IsOverriding}(a, V1) \mapsto \bot$
$\texttt{IsRecursiveMethod}(a, V1) \mapsto \bot$
$\texttt{IsRecursiveMethod}(a, V1) \mapsto \bot$
$\texttt{HasReturnType}(a, V1, V2) \mapsto \bot$
$\texttt{HasParameterType}(a, V1) \mapsto \bot$
$\texttt{HasParameterType}(V1, a) \mapsto \bot$
$\texttt{MethodHasParameterType}(a, V1, V2) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(a, V1, [V2], [V3]) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(V1, V2, [a], [a]) \mapsto \bot$
$\texttt{ExistsMethodDefinition}(a, m) \mapsto \bot$
$\texttt{ExistsMethodDefinition}(a, m1) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(V1, m, [V2]) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(V1, m1, [V2]) \mapsto \bot$

28 SpecialiseParameter

Rôle. SpecialiseParameter(classname s, subclasses [a,b], methodname m,paramType t ,paramName p, subtypes [st,q],new paramType st) : changer le type t du paramètre p des méthodes s : :m, a : :m et b : :m vers l'un de ses sous types (st).

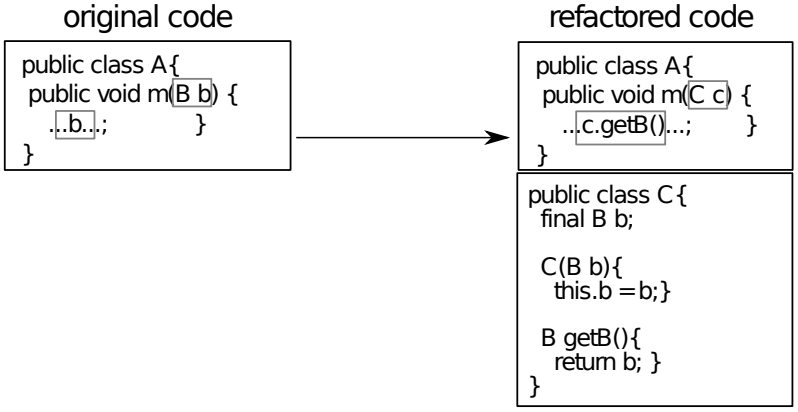
Précondition.
$(\texttt{IsSubType}(t1, t)$
$\wedge \neg\texttt{MethodIsUsedWithType}(s, m, [t], [t])$
$\wedge \neg\texttt{MethodIsUsedWithType}(a, m, [t], [t])$
$\wedge \neg\texttt{MethodIsUsedWithType}(b, m, [t], [t])$
$\wedge \texttt{ExistsClass}(s)$
$\wedge \texttt{ExistsMethodDefinitionWithParams}(s, m, [t])$
$\wedge \texttt{ExistsType}(t)$
$\wedge \texttt{ExistsType}(t1)$

$\wedge\ \neg\texttt{ExistsMethodDefinitionWithParams}(s, m, [t1])$
$\wedge\ \neg\texttt{IsInheritedMethodWithParams}(s, m, [t1])$
$\wedge\ \texttt{AllSubclasses}(s, [a; b])$
$\wedge\ \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(s, m, p)$
$\wedge\ (\neg\texttt{MethodIsUsedWithType}(s, m, [t], [t2])$
$\quad \vee\ \texttt{ExistsMethodDefinitionWithParams}(s, m, [t2])))$

Rétro-description.
$\texttt{ExistsMethodDefinitionWithParams}(s, m, [t1]) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(s, m, [t]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(a, m, [t]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(b, m, [t]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(a, m, [t1]) \mapsto \texttt{ExistsMethodDefinitionWithParams}(a, m, [t])$
$\texttt{ExistsMethodDefinitionWithParams}(b, m, [t1]) \mapsto \texttt{ExistsMethodDefinitionWithParams}(b, m, [t])$
$\texttt{MethodIsUsedWithType}(s, m, [t], [t]) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(a, m, [t], [t]) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(b, m, [t], [t]) \mapsto \bot$
$\texttt{ExistsParameterWithName}(s, m, [t1], p) \mapsto \top$
$\texttt{ExistsParameterWithName}(a, m, [t1], p) \mapsto \top$
$\texttt{ExistsParameterWithName}(b, m, [t1], p) \mapsto \top$
$\texttt{ExistsParameterWithType}(s, m, [t1], t1) \mapsto \top$
$\texttt{ExistsParameterWithType}(a, m, [t1], t1) \mapsto \top$
$\texttt{ExistsParameterWithType}(b, m, [t1], t1) \mapsto \top$
$\texttt{IsInverter}(s, m, t1, V) \mapsto \texttt{IsInverter}(s, m, t, V)$
$\texttt{IsInverter}(a, m, t1, V) \mapsto \texttt{IsInverter}(a, m, t, V)$
$\texttt{IsInverter}(b, m, t1, V) \mapsto \texttt{IsInverter}(b, m, t, V)$
$\texttt{IsInheritedMethodWithParams}(s, m, [t1]) \mapsto \top$
$\texttt{IsInheritedMethodWithParams}(a, m, [t1]) \mapsto \top$
$\texttt{IsInheritedMethodWithParams}(b, m, [t1]) \mapsto \top$
$\texttt{IsUsedConstructorAsMethodParameter}(t1, s, m) \mapsto \texttt{IsUsedConstructorAsMethodParameter}(t, s, m)$
$\texttt{IsUsedConstructorAsMethodParameter}(t1, a, m) \mapsto \texttt{IsUsedConstructorAsMethodParameter}(t, a, m)$
$\texttt{IsUsedConstructorAsMethodParameter}(t1, b, m) \mapsto \texttt{IsUsedConstructorAsMethodParameter}(t, b, m)$
$\texttt{IsOverridden}(a, m) \mapsto \texttt{ExistsMethodDefinition}(a, m)$
$\texttt{IsOverridden}(b, m) \mapsto \texttt{ExistsMethodDefinition}(b, m)$
$\texttt{IsOverriding}(a, m) \mapsto \texttt{ExistsMethodDefinition}(a, m)$
$\texttt{IsOverriding}(b, m) \mapsto \texttt{ExistsMethodDefinition}(b, m)$

29 IntroduceParameterObject

Rôle. introduceParmeterObject s [a,b] m [t1,t2] [p1,p2] t n créer une classe de type t, déplacer les paramètre p1 et p2 vers cette classe comme des variables d'instance et enfin changer m(t1 p1,t2 p2) vers m(t a). Les anciens accès vers p1 et p1 dans le corps de la méthode m sont remplacés désormais par a.p1 et a.p2.



Outils de refactoring. *Extract Parameter Object* avec IntelliJ IDEA et *Introduce Parameter Object* avec Eclipse.

Précondition.
$(\texttt{ExistsType}(s)$
$\wedge \neg\texttt{ExistsType}(t)$
$\wedge \texttt{ExistsMethodDefinitionWithParams}(s, m, [t1; t2])$
$\wedge \texttt{AllSubclasses}(s, [a; b])$
$\wedge \texttt{BoundVariableInMethodBody}(a, m, v1)$
$\wedge \texttt{BoundVariableInMethodBody}(a, m, v2)$
$\wedge \texttt{BoundVariableInMethodBody}(b, m, v1)$
$\wedge \texttt{BoundVariableInMethodBody}(b, m, v2))$

Rétro-description.
$\texttt{ExistsClass}(t) \mapsto \top$
$\texttt{ExistsType}(t) \mapsto \top$
$\texttt{IsPrimitiveType}(t) \mapsto \bot$
$\texttt{IsUsedMethod}(s, m, [t]) \mapsto \top$
$\texttt{IsUsedMethod}(s, m, [t1; t2]) \mapsto \bot$
$\texttt{IsUsedMethodIn}(t, C, M) \mapsto \bot$
$\texttt{IsUsedMethod}(t, C, [T1]) \mapsto \bot$
$\texttt{IsUsedMethod}(t, C, [T1; T2]) \mapsto \bot$
$\texttt{IsUsedMethod}(t, C, [T1; T2; T3]) \mapsto \bot$
$\texttt{IsUsedMethod}(t, C, [T1; T2; T3; T4]) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(t, C, m) \mapsto \bot$
$\texttt{IsUsedConstructorAsInitializer}(t, C, M) \mapsto \bot$
$\texttt{IsUsedConstructorAsMethodParameter}(C, t, M) \mapsto \bot$
$\texttt{IsUsedConstructorAsInitializer}(C, t, M) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(C, t, M) \mapsto \bot$
$\texttt{IsUsedConstructorAsInitializer}(t, C, M) \mapsto \bot$
$\texttt{IsSubType}(C, t) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(s, m, [t], [t]) \mapsto \top$
$\texttt{MethodIsUsedWithType}(s, m, [t1; t2], [t1; t2]) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(s, M, [t]) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(a, m, [t])$
$\quad \mapsto \texttt{IsInheritedMethodWithParams}(a, m, [t1; t2])$

$\texttt{IsInheritedMethodWithParams}(b, m, [t])$
$\quad \mapsto \texttt{IsInheritedMethodWithParams}(b, m, [t1; t2])$
$\texttt{IsInheritedMethod}(t, M)$
$\quad \mapsto \texttt{IsVisible}(java.lang.Object, M, t)$
$\texttt{IsInheritedMethodWithParams}(t, M, [\,])$
$\quad \mapsto \texttt{IsVisibleMethod}(java.lang.Object, M, [\,], t)$
$\texttt{IsInheritedMethodWithParams}(t, M, [T1])$
$\quad \mapsto \texttt{IsVisibleMethod}(java.lang.Object, M, [T1], t)$
$\texttt{IsInheritedMethodWithParams}(t, M, [T1; T2])$
$\quad \mapsto \texttt{IsVisibleMethod}(java.lang.Object, M, [T1; T2], t)$
$\texttt{IsInheritedMethodWithParams}(t, M, [T1; T2; T3])$
$\quad \mapsto \texttt{IsVisibleMethod}(java.lang.Object, M, [T1; T2; T3], t)$
$\texttt{IsInheritedMethodWithParams}(t, M, [T1; T2; T3; T4; T5])$
$\quad \mapsto \texttt{IsVisibleMethod}(java.lang.Object, M, [T1; T2; T3; T4; T5], t)$
$\texttt{ExistsMethodDefinitionWithParams}(t, M, [\,]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(t, M, [T1]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(t, M, [T1; T2]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(t, M, [T1; T2; T3]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(t, M, [T1; T2; T3; T4]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(t, M, [T1; T2; T3; T4; T5]) \mapsto \bot$
$\texttt{IsSubType}(t, X) \mapsto \bot(condition)$
$\texttt{ExtendsDirectly}(t, X) \mapsto \bot(condition)$
$\texttt{IsInheritedMethodWithParams}(C, M, [t; T1]) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(C, M, [t]) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(C, M, [t; T1]) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(C, M, [T1; t]) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(C, M, [T1; T2; t]) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(C, M, [T1; t; T2]) \mapsto \bot$
$\texttt{IsInheritedMethodWithParams}(C, M, [t; T1; T2]) \mapsto \bot$
$\texttt{ExtendsDirectly}(t, java.lang.Object) \mapsto \top$
$\texttt{IsUsedConstructorAsObjectReceiver}(t, a, m) \mapsto \texttt{BoundVariableInMethodBody}(a, m, v1)$
$\texttt{IsUsedConstructorAsObjectReceiver}(t, a, m) \mapsto \texttt{BoundVariableInMethodBody}(a, m, v2)$
$\texttt{IsUsedConstructorAsObjectReceiver}(t, b, m) \mapsto \texttt{BoundVariableInMethodBody}(b, m, v1)$
$\texttt{IsUsedConstructorAsObjectReceiver}(t, b, m) \mapsto \texttt{BoundVariableInMethodBody}(b, m, v2)$
$\texttt{ExistsParameterWithName}(s, m, [t], n) \mapsto \top$
$\texttt{ExistsParameterWithName}(a, m, [t], n) \mapsto \top$
$\texttt{ExistsParameterWithName}(b, m, [t], n) \mapsto \top$
$\texttt{ExistsParameterWithType}(s, m, [t], t) \mapsto \top$
$\texttt{ExistsParameterWithType}(a, m, [t], t) \mapsto \top$
$\texttt{ExistsParameterWithType}(b, m, [t], t) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(s, m, [t]) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(C, M, [t]) \mapsto \bot(condition)$
$\texttt{ExistsMethodDefinitionWithParams}(C, M, [t; T1]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(C, M, [T1; t]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(C, M, [T1; T2; t]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(C, M, [T1; t; T2]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(C, M, [t; T1; T2]) \mapsto \bot$

$\texttt{ExistsMethodDefinitionWithParams}(s, m, [t1; t2]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(a, m, [t1; t2]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(b, m, [t1; t2]) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(a, m, [t]) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(b, m, [t]) \mapsto \top$
$\texttt{ExistsField}(t, p1) \mapsto \top$
$\texttt{ExistsField}(t, p2) \mapsto \top$
$\texttt{ExistsMethodDefinition}(t, getp1) \mapsto \top$
$\texttt{ExistsMethodDefinition}(t, getp2) \mapsto \top$
$\texttt{IsUsedConstructorAsObjectReceiver}(T1, T2, getp1) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(T1, T2, getp2) \mapsto \bot$
$\texttt{IsPrivate}(t, p1) \mapsto \top$
$\texttt{IsPrivate}(t, p2) \mapsto \top$
$\texttt{IsInheritedField}(t, p1) \mapsto \bot$
$\texttt{IsInheritedField}(t, p2) \mapsto \bot$
$\texttt{ExistsParameterWithName}(s, m, [t1; t2], p1) \mapsto \bot$
$\texttt{ExistsParameterWithName}(s, m, [t1; t2], p2) \mapsto \bot$
$\texttt{ExistsParameterWithName}(a, m, [t1; t2], p1) \mapsto \bot$
$\texttt{ExistsParameterWithName}(a, m, [t1; t2], p2) \mapsto \bot$
$\texttt{ExistsParameterWithName}(b, m, [t1; t2], p1) \mapsto \bot$
$\texttt{ExistsParameterWithName}(b, m, [t1; t2], p2) \mapsto \bot$
$\texttt{ExistsParameterWithType}(s, m, [t1; t2], t1) \mapsto \bot$
$\texttt{ExistsParameterWithType}(s, m, [t1; t2], t2) \mapsto \bot$
$\texttt{ExistsParameterWithType}(a, m, [t1; t2], t1) \mapsto \bot$
$\texttt{ExistsParameterWithType}(a, m, [t1; t2], t2) \mapsto \bot$
$\texttt{ExistsParameterWithType}(b, m, [t1; t2], t1) \mapsto \bot$
$\texttt{ExistsParameterWithType}(b, m, [t1; t2], t2) \mapsto \bot$
$\texttt{BoundVariableInMethodBody}(s, m, n) \mapsto \texttt{BoundVariableInMethodBody}(s, m, p1)$
$\texttt{BoundVariableInMethodBody}(s, m, n) \mapsto \texttt{BoundVariableInMethodBody}(s, m, p2)$
$\texttt{BoundVariableInMethodBody}(s, m, n) \mapsto \texttt{BoundVariableInMethodBody}(s, m, p1)$
$\texttt{BoundVariableInMethodBody}(s, m, n) \mapsto \texttt{BoundVariableInMethodBody}(s, m, p2)$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(s, m, n, T) \mapsto \top$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(a, m, n, T) \mapsto \top$
$\texttt{AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC}(b, m, n, T) \mapsto \top$
$\texttt{ExistsParameterWithName}(s, m, [t1; t2; t], n) \mapsto \top$
$\texttt{ExistsParameterWithName}(a, m, [t1; t2; t], n) \mapsto \top$
$\texttt{ExistsParameterWithName}(b, m, [t1; t2; t], n) \mapsto \top$
$\texttt{ExistsParameterWithType}(s, m, [t1; t2; t], t) \mapsto \top$
$\texttt{ExistsParameterWithType}(a, m, [t1; t2; t], t) \mapsto \top$
$\texttt{ExistsParameterWithType}(b, m, [t1; t2; t], t) \mapsto \top$

30 DeleteDuplicateMethod

Rôle. DeleteDuplicateMethod c m [t1,t2] s : supprimer la méthode c : :m qui est équivalente de point de vue sémantique à la méthode s : :m qui est héritée par c.

Outils de refactoring. *Delete* avec Eclipse et IntelliJ IDEA.

Précondition.
$\texttt{ExistsClass}(c)$
$\wedge\, \texttt{ExistsType}(c)$
$\wedge\, \texttt{ExistsMethodDefinitionWithParams}(c, m, [t1; t2])$
$\wedge\, \texttt{IsInheritedMethod}(c, m)$
$\wedge\, \texttt{IsDuplicate}(c, m, s, m)$
$\wedge\, \texttt{AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded}(c, m, this))$

Rétro-description.
$\texttt{ExistsParameterWithType}(c, m, [t1; t2], V1) \mapsto \bot$
$\texttt{ExistsMethodDefinitionWithParams}(c, m, [t1; t2]) \mapsto \bot$
$\texttt{ExistsMethodDefinition}(c, m) \mapsto \bot$
$\texttt{BoundVariableInMethodBody}(c, m, t1) \mapsto \bot$
$\texttt{BoundVariableInMethodBody}(c, m, t2) \mapsto \bot$
$\texttt{ExistsParameterWithName}(c, m, [t1; t2], V1) \mapsto \bot$
$\texttt{IsIndirectlyRecursive}(c, m) \mapsto \bot$
$\texttt{IsVisibleMethod}(c, m, [t1; t2], V1) \mapsto \bot$
$\texttt{IsInverter}(c, m, V1, V2) \mapsto \bot$
$\texttt{IsDelegator}(c, V1, m) \mapsto \bot$
$\texttt{IsOverridden}(c, m) \mapsto \bot$
$\texttt{IsOverloaded}(c, m) \mapsto \bot$
$\texttt{IsOverriding}(c, m) \mapsto \bot$
$\texttt{IsRecursiveMethod}(c, m) \mapsto \bot$
$\texttt{HasReturnType}(c, m, V1) \mapsto \bot$
$\texttt{MethodHasParameterType}(c, m, V1) \mapsto \bot$
$\texttt{MethodIsUsedWithType}(c, m, [t1; t2], [t1; t2]) \mapsto \bot$

31 DuplicateMethodInHierarchyGen

Rôle. DuplicateMethodInHierarchyGen (class name s, sublclasslist [a ;b], methodname m, return types [r1 ;r2], invokedmethodsInmethodname [m1 ;m2], callermethods [m3 ;m4], newname n, methodnamparameters [t1 ;t2] :dupliquer la méthode s : :m dont son type de retour est un type générique vers deux méthodes avec le nom n qui auront respectivement les types de retour r1 et r2.

Outils de refactoring. La même démarche que celle de l'opération *DuplicateMethodInHierarchy* sauf qu'on précise le type de retour pour chaque nouvelle méthode avec l'opération *change signature*.

Précondition.
$(\texttt{ExistsClass}(s)$
$\wedge\, \texttt{ExistsMethodDefinitionWithParams}(s, m, [t1; t2])$
$\wedge\, \texttt{ExistsMethodDefinition}(s, m)$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(s, n, [t1; t2])$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(a, n, [t1; t2])$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(b, n, [t1; t2])$

$\wedge$ ExistsType$(r1)$
$\wedge$ ExistsType$(r2)$
$\wedge \neg$IsInheritedMethodWithParams$(s, n, [t1; t2])$
$\wedge$ AllSubclasses$(s, [a; b])$)

Rétro-description.

ExistsMethodDefinition$(s, n) \mapsto \top$

ExistsMethodDefinitionWithParams$(s, n, [t1; t2]) \mapsto \top$

AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded$(s, n, V) \mapsto \top(condition)$

AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC$(s, n, V, V1)$
$\mapsto$ AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC$(s, m, V, V1)$

BoundVariableInMethodBody(s, n, V)
$\mapsto$ BoundVariableInMethodBody(s, m, V)

IsPublic$(s, n) \mapsto$ IsPublic(s, m)

ExistsParameterWithName$(s, n, [t1; t2], V)$
$\mapsto$ ExistsParameterWithName$(s, m, [t1; t2], V)$

ExistsParameterWithType$(s, n, [t1; t2], T)$
$\mapsto$ ExistsParameterWithType$(s, m, [t1; t2], T)$

IsIndirectlyRecursive(s, n)
$\mapsto$ IsIndirectlyRecursive(s, m)

IsRecursiveMethod(s, n)
$\mapsto$ IsRecursiveMethod(s, m)

IsInverter(s, n, T, V)
$\mapsto$ IsInverter(s, m, T, V)

IsUsedAttributeInMethodBody(s, V, n)
$\mapsto$ IsUsedAttributeInMethodBody(s, V, m)

MethodHasParameterType(s, n, V)
$\mapsto$ MethodHasParameterType(s, m, V)

ExistsMethodDefinitionWithParams$(a, n, [t1; t2])$
$\mapsto$ ExistsMethodDefinitionWithParams$(a, m, [t1; t2])$

ExistsMethodDefinitionWithParams$(b, n, [t1; t2])$
$\mapsto$ ExistsMethodDefinitionWithParams$(b, m, [t1; t2])$

IsDelegator$(s, n, m3) \mapsto \top$

IsDelegator$(s, n, m4) \mapsto \top$

IsDelegator$(a, n, m3) \mapsto \top$

IsDelegator$(a, n, m4) \mapsto \top$

IsDelegator$(b, n, m3) \mapsto \top$

IsDelegator$(b, n, m4) \mapsto \top$

ExistsMethodDefinition$(s, n) \mapsto \top$

ExistsMethodDefinition$(a, n) \mapsto \top$

ExistsMethodDefinition$(b, n) \mapsto \top$

MethodIsUsedWithType$(s, n, [t1; t2], [t1; t2]) \mapsto \bot$

MethodIsUsedWithType$(a, n, [t1; t2], [t1; t2]) \mapsto \bot$

MethodIsUsedWithType$(b, n, [t1; t2], [t1; t2]) \mapsto \bot$

MethodIsUsedWithType$(s, n, [t1; t2], [T]) \mapsto \bot$

MethodIsUsedWithType$(a, n, [t1; t2], [T]) \mapsto \bot$

MethodIsUsedWithType$(b, n, [t1; t2], [T]) \mapsto \bot$

$\texttt{ExistsMethodInvocation}(s, m1, V, n) \mapsto \top$
$\texttt{ExistsMethodInvocation}(s, m2, V, n) \mapsto \top$
$\texttt{ExistsMethodInvocation}(a, m1, V, n) \mapsto \top$
$\texttt{ExistsMethodInvocation}(a, m2, V, n) \mapsto \top$
$\texttt{ExistsMethodInvocation}(b, m1, V, n) \mapsto \top$
$\texttt{ExistsMethodInvocation}(b, m2, V, n) \mapsto \top$
$\texttt{IsInheritedMethodWithParams}(a, n, [t1; t2]) \mapsto \neg\texttt{ExistsMethodDefinitionWithParams}(a, m, [t1; t2])$
$\texttt{IsInheritedMethodWithParams}(b, n, [t1; t2]) \mapsto \neg\texttt{ExistsMethodDefinitionWithParams}(b, m, [t1; t2])$
$\texttt{IsInheritedMethod}(a, n) \mapsto \neg\texttt{ExistsMethodDefinition}(a, m)$
$\texttt{IsInheritedMethod}(b, n) \mapsto \neg\texttt{ExistsMethodDefinition}(b, m)$
$\texttt{IsOverriding}(a, n) \mapsto \neg\texttt{ExistsMethodDefinition}(a, m)$
$\texttt{IsOverriding}(b, n) \mapsto \neg\texttt{ExistsMethodDefinition}(b, m)$
$\texttt{IsOverridden}(a, n) \mapsto \neg\texttt{ExistsMethodDefinition}(a, m)$
$\texttt{IsOverridden}(b, n) \mapsto \neg\texttt{ExistsMethodDefinition}(b, m)$

32 AddSpecializedMethodInHierarchyGen (composée)

Rôle. AddSpecializedMethodInHierarchyGen(class s, subclasses [a,b], methodname m, returntypes [r1,r2], callermethods [n,o], inkvokedmethods [p,q], paramtype t, paramname pn , subtypesOfparamtype [t1,t2],newtype t') : cette opération est similaire à l'opération AddSpecializedMethodInHierarchy 20 sauf qu'elle est appliquée sur des méthodes dont leurs types de retour sont des types génériques.

Algorithm of the operation L'opération AddSpecializedMethodInHierarchyGen est basée sur trois étapes :

AddSpecializedMethodInHierarchyGen(class s, subclasses [a,b], methodname m, returntypes [r1,r2], callermethods [n,o], inkvokedmethods [p,q], paramtype t, paramname pn , subtypesOfparamtype [t1,t2],newtype t') =

1. DuplicateMethodInHierarchyGen s [a,b] m [r1,r2] [p,q] [n,o] temporaryName [t]
2. SpecialiseParameter s [a,b] temporaryName t pn [t1,t2] t' ;
3. RenameDelegatorWithOverloading (s, [a,b], temporaryName,t', pn,t,m)

33 InlineConstructor

Rôle. InlineConstructor (classname s, methodname m, inlinedConstructor c, fields [f1,f2], getters [g1,g2] : Cette opérations est utilisée pour faire un *inline* pour le constructeur c qui est utilisé dans la méthode s : :m.

Outils de refactoring. *inline* avec IntelliJ IDEA.

Précondition.
$(\texttt{ExistsType}(s)$
$\wedge\ \texttt{ExistsType}(c)$
$\wedge\ \texttt{IsUsedConstructorAsObjectReceiver}(c, s, m)$

$\wedge$ ($\neg$IsInheritedMethodWithParams$(s, m, [c])$
$\vee \neg$ExistsMethodDefinitionWithParams$(s, m, [c])$))

Rétro-description.
IsUsedConstructorAsObjectReceiver$(c, s, m) \mapsto \bot$
ExistsMethodDefinitionWithParams$(C, M, [c]) \mapsto \bot$
IsUsedMethodIn$(c, c, s) \mapsto \bot$
IsUsedConstructorAsMethodParameter$(c, s, m) \mapsto \bot$
IsUsedConstructorAsInitializer$(c, s, m) \mapsto \bot$
existsFieldInMethodScope$(s, m, f1) \mapsto \top$
existsFieldInMethodScope$(s, m, f2) \mapsto \top$
BoundVariableInMethodBody$(s, m, f1) \mapsto \top$
BoundVariableInMethodBody$(s, m, f2) \mapsto \top$
existslocalVariable$(s, m, f1var) \mapsto \top$
existslocalVariable$(s, m, f2var) \mapsto \top$
ExistsMethodDefinition$(s, g1) \mapsto \top$
ExistsMethodDefinition$(s, g2) \mapsto \top$
IsUsedConstructorAsObjectReceiver$(c, s, g1) \mapsto \bot$
IsUsedConstructorAsObjectReceiver$(c, s, g2) \mapsto \bot$
IsOverriding$(s, g1) \mapsto \bot$
IsOverriding$(s, g2) \mapsto \bot$
IsOverridden$(s, g1) \mapsto \bot$
IsOverridden$(s, g2) \mapsto \bot$
IsRecursiveMethod$(s, g1) \mapsto \bot$
IsRecursiveMethod$(s, g2) \mapsto \bot$

34 InlineLocalField

Rôle. inlineLocalField (classname s, methodname m, fieldname f) : faire un *inline* pour le champs f qui est utilisé dans la portée de la méthode s : :m.

Outils de refactoring. *inline* avec Eclipse et IntelliJ IDEA.

Précondition.
(ExistsType(s)
$\wedge$ ExistsMethodDefinition(s, m)
$\wedge$ existsFieldInMethodScope(s, m, f))

Rétro-description.
existsFieldInMethodScope$(s, m, f) \mapsto \bot$

35 InlinelocalVariable

Rôle. InlinelocalVariable (classname s, methodname m, variablename v) : faire un *inline* pour la variable locale déclarée dans la portée de la méthode s : :m.

Outils de refactoring. *inline* avec Eclipse et IntelliJ IDEA.

Précondition.
$(\texttt{ExistsType}(s)$
$\wedge\, \texttt{ExistsMethodDefinition}(s, m)$
$\wedge\, \texttt{existslocalVariable}(s, m, v))$

Rétro-description. $\texttt{existslocalVariable}(s, m, v) \mapsto \bot$

36 InlineParmeterObject (composée)

Rôle. InlineParmeterObject (classname s, methodname m, inlinedConstructor c, inlinedgetters [g1,g2], fields [f1,f2] : faire un *inline* pour un objet paramètre (*parameter object*). Elle a le rôle inverse de l'opération IntroduceParameterObject.

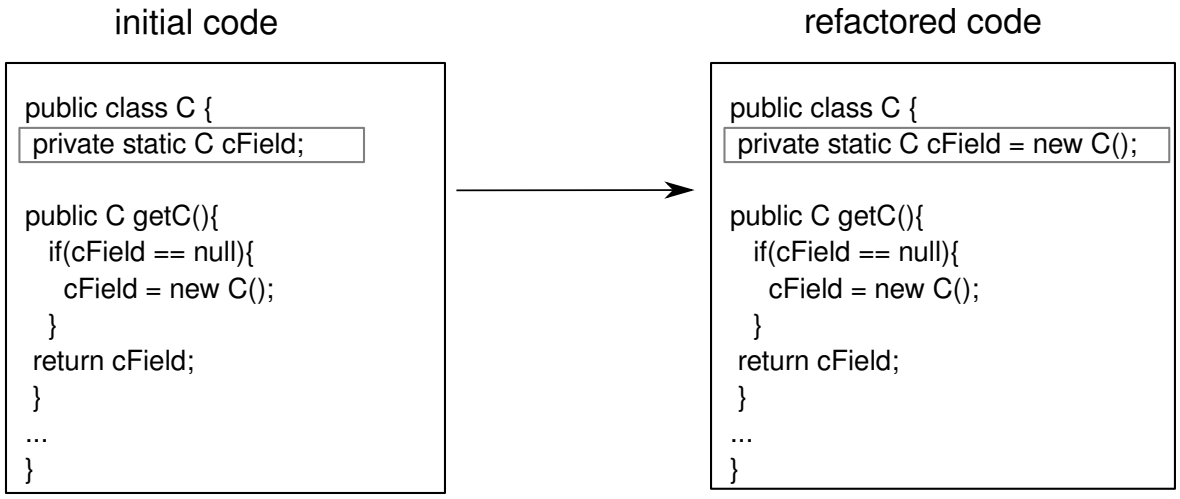
Algorithm of the operation L'algorithme suivant décrit la séquence d'opérations de refactoring utilisées pour appliquer cette opération :

InlineParmeterObject (s, m, c, [g1,g2], [f1,f2]) =

1. InlineConstructor s m c [g1,g2] [f1,f2]
2. Forall f in [f1,f2] do InlineLocalField s m f
3. Forall g in [g1,g2] do InlineAnddelete (s,g,[], m, [], [])
4. Forall f in [f1,f2] do InlinelocalVariable s m (generate variable name from f)

37 InitializeStaticField

Rôle. InitializeStaticField(c,f,t,v) : cette opération sert à initialiser le champs $c :: f$ qui est de type t par la valeur v. Cette opération est à usage restrictive et elle est conçue pour traiter des cas très spécifiques.



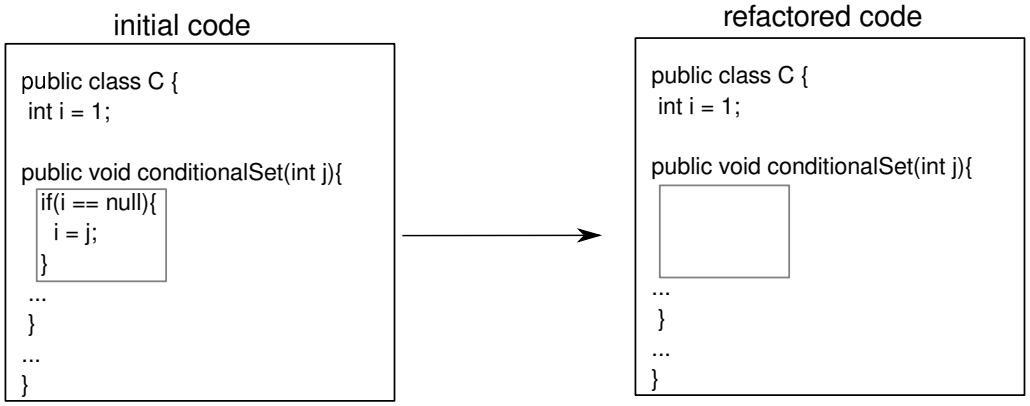
Outils de refactoring. Cette opération n'est pas encore implémentée.

Précondition.
$\texttt{ExistsClass}(c)$
$\wedge\, \texttt{ExistsField}(c, f)$
$\wedge\, \texttt{IsPrivate}(c, f)$
$\wedge\, \texttt{HasType}(v, t)$
$\wedge\, \neg\texttt{IsInitialized}(c, f)$
$\wedge\, \texttt{IsStatic}(c, f)$
$\wedge\, (\neg\texttt{IsFieldValueModified}(c, f)$
$\quad \vee\, \texttt{FieldhasValue}(c, f, v))$

Rétro-description.
$\texttt{IsInitialized}(c, f) \mapsto \top$
$\texttt{FieldhasValue}(c, f, v) \mapsto \top$
$\texttt{IsTrue}(c, M, C, (f == null)) \mapsto \bot$

38 DeleteUnusedConditionalStructure

Rôle DeleteUnusedConditionalStructure(c,m,cs,cn) : cette opération est utilisée pour supprimer la structure conditionnelle cs qui se trouve dans le corps de la méthode $c :: m$ et dont sa condition est cn. La structure conditionnelle à supprimer n'est pas utilisée et n'a aucun effet sur le comportement du programme et donc sa suppression ne viole pas la sémantique



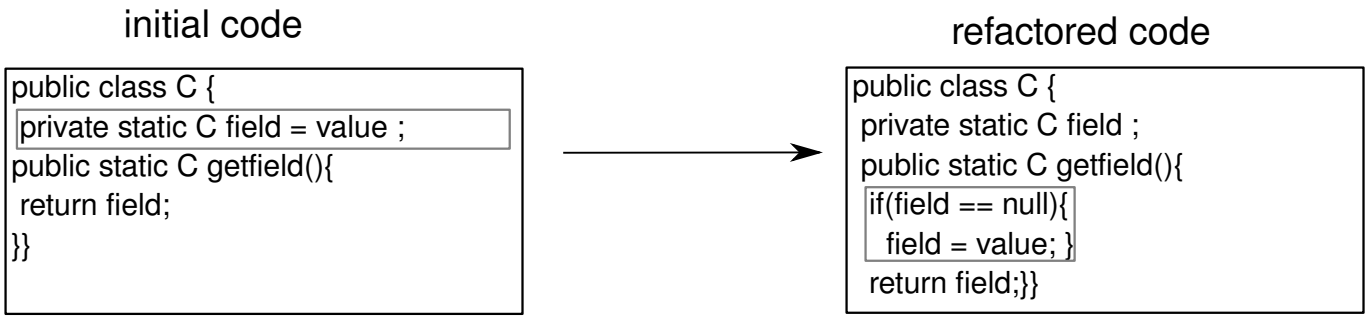
Outils de refactoring. Cette opération n'est pas encore implémentée.

Précondition.
$\texttt{ExistsClass}(c)$
$\wedge\, \texttt{ExistsMethodDefinition}(c, m)$
$\wedge\, \texttt{ExistsCode}(c, m, cs)$
$\wedge\, \neg\texttt{isConditionSatistfied}(c, m, cs, cn)$

Rétro-description.
$\texttt{ExistsCode}(c, m, cs) \mapsto \bot$

39 CreateLazyLoading

Rôle L'opération CreateLazyLoading(c,m,f,t,v) est utilisée pour annuler l'initialisation du champs $c :: f$ par la valeur v au moment du chargement du programme et de créer une initialisation paresseuse dans le corps de la méthode $c :: m$ et qui sera déclenchée une seule fois lors du premier appel de cette méthode.



Outils de refactoring. Cette opération n'est pas encore implémentée.

Précondition.
$\texttt{ExistsClass}(c)$
$\wedge\, \texttt{ExistsMethodDefinition}(c, m)$
$\wedge\, \texttt{ExistsField}(c, f)$
$\wedge\, \texttt{IsStatic}(c, f)$
$\wedge\, \texttt{IsInitialized}(c, f)$
$\wedge\, \texttt{IsPrivate}(c, f)$
$\wedge\, \texttt{FieldhasValue}(c, f, v)$

Rétro-description.
$\texttt{ExistsCode}(c, m, if(f == null)f = v;) \mapsto \top$
$\texttt{FieldhasValue}(c, f, null) \mapsto \top$
$\texttt{IsInitialized}(c, f) \mapsto \bot$

40 MoveStaticMethod

L'opération moveStaticMethod (a,[f1 ;f2],m,[t1 ;t2],b) est utilisée pour déplacer statique.

Outils de refactoring. Opération *Move* de IntelliJ IDEA ou Eclipse.

Précondition.
$\texttt{ExistsClass}(a)$
$\wedge\, \texttt{ExistsClass}(b)$
$\wedge\, \texttt{ExistsMethodDefinition}(a, m)$
$\wedge\, \texttt{ExistsMethodDefinitionWithParams}(a, m, [t1; t2])$
$\wedge\, \neg\texttt{ExistsMethodDefinitionWithParams}(b, m, [t1; t2])$
$\wedge\, \texttt{IsStatic}(a, m)$
$\wedge\, \neg\texttt{IsPrivate}(a, m)$
$\wedge\, (\neg\texttt{BoundVariableInMethodBody}(a, m, f1)$
$\quad \vee\, (\neg\texttt{IsPrivate}(a, f1)$
$\wedge\, \texttt{IsStatic}(a, f1)))$
$\wedge\, (\neg\texttt{BoundVariableInMethodBody}(a, m, f2)$
$\quad \vee\, (\neg\texttt{IsPrivate}(a, f2)$
$\wedge\, \texttt{IsStatic}(a, f2)))$

Rétro-description. $\texttt{ExistsMethodDefinitionWithParams}(a, m, [t1; t2]) \mapsto \bot$
$\texttt{ExistsMethodDefinition}(a, m) \mapsto \bot$
$\texttt{ExistsMethodDefinition}(b, m) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(b, m, [t1; t2]) \mapsto \top$
$\texttt{BoundVariableInMethodBody}(b, m, M) \mapsto \texttt{BoundVariableInMethodBody}(a, m, M)$
$\texttt{IsUsedMethodIn}(b, m, b) \mapsto \texttt{IsUsedMethodIn}(a, m, b)$

41 MakeMethodVisibilityPublic

L'opération makeMethodVisibilityPublic (c,m) est utilisée pour rendre la visibilité d'une méthode ou d'un constructeur publique.

Outils de refactoring. Opération *change signature* de IntelliJ IDEA ou Eclipse.

Précondition.
$\texttt{ExistsClass}(c)$
$\wedge\, \texttt{ExistsMethodDefinition}(c, m)$
$\wedge\, \texttt{IsPrivate}(c, m))$

Rétro-description.
$\texttt{IsPrivate}(c, m) \mapsto \bot$
$\texttt{IsPublic}(c, m) \mapsto \top$

42 ReplaceConstructorWithMethodFactory

L'opération replaceConstructorWithMethodFactory (s,c,[t1 ;t2],m) est utilisée pour créer une méthode m qui remplace le rôle du constructeur c de la classe a. Après l'application de cette opération, l'instanciation de la classe a se fait par l'intermédiaire de la méthode m.

Outils de refactoring. Opération *Replace Constructor With Factory Method* de IntelliJ IDEA ou Eclipse.

Précondition.
$\texttt{ExistsClass}(s)$
$\wedge\, \texttt{ExistsMethodDefinitionWithParams}(s, c, [t1; t2])$
$\wedge\, \neg\texttt{ExistsMethodDefinition}(s, m)$

Rétro-description.
$\texttt{ExistsMethodDefinition}(s, m) \mapsto \top$
$\texttt{ExistsMethodDefinitionWithParams}(s, m, [t1; t2]) \mapsto \top$
$\texttt{IsPrivate}(s, c) \mapsto \top$
$\texttt{IsStatic}(s, m) \mapsto \top$
$\texttt{HasReturnType}(s, m, s) \mapsto \top$
$\texttt{IsUsedConstructorAsObjectReceiver}(c, s, m) \mapsto \top$
$\texttt{IsUsedConstructorAsMethodParameter}(c, C, M) \mapsto \bot$
$\texttt{IsUsedConstructorAsInitializer}(c, C, M) \mapsto \bot$
$\texttt{IsUsedConstructorAsObjectReceiver}(c, C, M) \mapsto \bot$

C

Préconditions de JHotDraw

1 Chaines d'opérations pour un une transformation aller-retour appliquée sur l'instance Composite de JhotDraw

Le nombre total de ces chaines d'opérations est de l'ordre de 944 opérations. Nous présentons ici quelques une pour avoir une idée sur la séquence des opérations composant la transformation de JHotDraw.
PushDownCopy(LineConnectionFigure, BezierFigure, [findFigureInside ; setAttribute ; contains]) ;
PushDownCopy(SVGPath, AbstractCompositeFigure, [addNotify ; removeNotify ; findFigureInside ; contains]) ;
PushDownCopy(LabeledLineConnectionFigure, BezierFigure, [basicTransform ; setAttribute ; findFigureInside ; contains]) ;
PushDownCopy(DependencyFigure, LineConnectionFigure, [addNotify ; basicTransform ; setAttribute ; findFigureInside ; contains ;]) ;
PushDownCopy(NodeFigure, TextFigure, [addNotify ; basicTransform ; setAttribute ; findFigureInside ; contains]) ;
PushDownCopy(GraphicalCompositeFigure, AbstractCompositeFigure, [findFigureInside]) ;
Createindirectioninsuperclass(AbstractFigure, [LabeledLineConnectionFigure ; AbstractCompositeFigure ; GraphicalCompositeFigure ; EllipseFigure ; DiamondFigure ; RectangleFigure ; RoundRectangleFigure ; TriangleFigure ; TextFigure ; BezierFigure ; TextAreaFigure ;
NodeFigure ; SVGImage ; SVGPath ; DependencyFigure ; LineConnectionFigure],
basicTransform, [AffineTransform tx], Void, basicTransformTmpVC) ;
Createindirectioninsuperclass(AbstractFigure, [LabeledLineConnectionFigure ; AbstractCompositeFigure ; GraphicalCompositeFigure ; EllipseFigure ; DiamondFigure ; RectangleFigure ; RoundRectangleFigure ; TriangleFigure ; TextFigure ; BezierFigure ; TextAreaFigure ; NodeFigure ; SVGImage ; SVGPath ; DependencyFigure ; LineConnectionFigure], contains, [Point2D.Double p], Boolean, containsTmpVC) ;
Createindirectioninsuperclass(AbstractFigure, [LabeledLineConnectionFigure ; AbstractCompositeFigure ; GraphicalCompositeFigure ; EllipseFigure ; DiamondFigure ; RectangleFigure ; RoundRectangleFigure ; TriangleFigure ; TextFigure ; BezierFigure ; TextAreaFigure ; NodeFigure ; SVGImage ; SVGPath ; DependencyFigure ; LineConnectionFigure], setAttribute, [AttributeKey key ; Object value], Void, setAttributeTmpVC) ;
Createindirectioninsuperclass(AbstractFigure, [LabeledLineConnectionFigure ; AbstractCompositeFigure ; GraphicalCompositeFigure ; EllipseFigure ; DiamondFigure ; RectangleFigure ; RoundRectangleFigure ; TriangleFigure ; TextFigure ; BezierFigure ; TextAreaFigure ; NodeFigure ; SVGImage ; SVGPath ; DependencyFigure ; LineConnectionFigure], findFigureInside, [Point2D.Double p], Figure, findFigureInsideTmpVC) ;
Createindirectioninsuperclass(AbstractFigure, [LabeledLineConnectionFigure ; AbstractCompositeFigure ; GraphicalCompositeFigure ; EllipseFigure ; DiamondFigure ; RectangleFigure ; RoundRectangleFigure ; TriangleFigure ; TextFigure ; BezierFigure ; TextAreaFigure ; NodeFigure ; SVGImage ; SVGPath ; DependencyFigure ; LineConnectionFigure], addNotify, [Drawing d], Void, addNotifyTmpVC) ;
Createindirectioninsuperclass(AbstractFigure, [LabeledLineConnectionFigure ; AbstractCompositeFigure ; GraphicalCompositeFigure ; EllipseFigure ; DiamondFigure ; RectangleFigure ; RoundRectangleFigure ; TriangleFigure ; TextFigure ; BezierFigure ; TextAreaFigure ; NodeFigure ; SVGImage ; SVGPath ; DependencyFigure ; LineConnectionFigure], removeNotify, [Drawing d], Void, removeNotifyTmpVC) ;
introduceParmeterObject(AbstractFigure, [LabeledLineConnectionFigure ; AbstractCompositeFigure ; GraphicalCompositeFigure ; EllipseFigure ; DiamondFigure ; RectangleFigure ; RoundRectangleFigure ; TriangleFigure ; TextFigure ; BezierFigure ; TextAreaFigure ; NodeFigure ; SVGImage ; SVGPath ; DependencyFigure ; LineConnectionFigure], ba-

sicTransformTmpVC, [AffineTransform tx], [AffineTransform tx], [tx], BasicTransformVisitor, v) ;
introduceParmeterObject(AbstractFigure, [LabeledLineConnectionFigure ; AbstractCompositeFigure ; GraphicalCompositeFigure ; EllipseFigure ; DiamondFigure ; RectangleFigure ; RoundRectangleFigure ; TriangleFigure ; TextFigure ; BezierFigure ; TextAreaFigure ; NodeFigure ; SVGImage ; SVGPath ; DependencyFigure ; LineConnectionFigure], containsTmpV
[Point2D.Double p], [Point2D.Double p], [p], ContainsVisitor, v) ;
introduceParmeterObject(AbstractFigure, [LabeledLineConnectionFigure ; AbstractCompositeFigure ; GraphicalCompositeFigure ; EllipseFigure ; DiamondFigure ; RectangleFigure ; RoundRectangleFigure ; TriangleFigure ; TextFigure ; BezierFigure ; TextAreaFigure ; NodeFigure ; SVGImage ; SVGPath ; DependencyFigure ; LineConnectionFigure], setAttributeTmpVC, [AttributeKey key ; Object value], [AttributeKey key ; Object value], [key ; value], SetAttributeVisitor, v) ;
introduceParmeterObject(AbstractFigure, [LabeledLineConnectionFigure ; AbstractCompositeFigure ; GraphicalCompositeFigure ; EllipseFigure ; DiamondFigure ; RectangleFigure ; RoundRectangleFigure ; TriangleFigure ; TextFigure ; BezierFigure ; TextAreaFigure ; NodeFigure ; SVGImage ; SVGPath ; DependencyFigure ; LineConnectionFigure], findFigureInsideTmpVC, [Point2D.Double p], [Point2D.Double p], [p], FindFigureInsideVisitor, v) ;
introduceParmeterObject(AbstractFigure, [LabeledLineConnectionFigure ; AbstractCompositeFigure ; GraphicalCompositeFigure ; EllipseFigure ; DiamondFigure ; RectangleFigure ; RoundRectangleFigure ; TriangleFigure ; TextFigure ; BezierFigure ; TextAreaFigure ; NodeFigure ; SVGImage ; SVGPath ; DependencyFigure ; LineConnectionFigure], addNotifyTmpVC, [Drawing d], [Drawing d], [d], AddNotifyVisitor, v) ;
introduceParmeterObject(AbstractFigure, [LabeledLineConnectionFigure ; AbstractCompositeFigure ; GraphicalCompositeFigure ; EllipseFigure ; DiamondFigure ; RectangleFigure ; RoundRectangleFigure ; TriangleFigure ; TextFigure ; BezierFigure ; TextAreaFigure ; NodeFigure ; SVGImage ; SVGPath ; DependencyFigure ; LineConnectionFigure], removeNotifyTmpVC, [Drawing d], [Drawing d], [d], RemoveNotifyVisitor, v) ;
...

2 Précondition générée

$$
\begin{aligned}
&\texttt{IsInheritedMethod}(GraphicalCompositeFigure, findFigureInside)\\
&\wedge\ \texttt{IsInheritedMethod}(NodeFigure, contains)\\
&\wedge\ \texttt{IsInheritedMethod}(NodeFigure, findFigureInside)\\
&\wedge\ \texttt{IsInheritedMethod}(NodeFigure, setAttribute)\\
&\wedge\ \texttt{IsInheritedMethod}(NodeFigure, basicTransform)\\
&\wedge\ \texttt{IsInheritedMethod}(NodeFigure, addNotify)\\
&\wedge\ \texttt{IsInheritedMethod}(DependencyFigure, contains)\\
&\wedge\ \texttt{IsInheritedMethod}(DependencyFigure, findFigureInside)\\
&\wedge\ \texttt{IsInheritedMethod}(DependencyFigure, setAttribute)\\
&\wedge\ \texttt{IsInheritedMethod}(DependencyFigure, basicTransform)\\
&\wedge\ \texttt{IsInheritedMethod}(DependencyFigure, addNotify)\\
&\wedge\ \texttt{IsInheritedMethod}(LabeledLineConnectionFigure, contains)\\
&\wedge\ \texttt{IsInheritedMethod}(LabeledLineConnectionFigure, findFigureInside)\\
&\wedge\ \texttt{IsInheritedMethod}(LabeledLineConnectionFigure, setAttribute)\\
&\wedge\ \texttt{IsInheritedMethod}(LabeledLineConnectionFigure, basicTransform)\\
&\wedge\ \texttt{IsInheritedMethod}(SVGPath, contains)\\
&\wedge\ \texttt{IsInheritedMethod}(SVGPath, findFigureInside)\\
&\wedge\ \texttt{IsInheritedMethod}(SVGPath, removeNotify)\\
&\wedge\ \texttt{IsInheritedMethod}(SVGPath, addNotify)\\
&\wedge\ \texttt{IsInheritedMethod}(LineConnectionFigure, contains)\\
&\wedge\ \texttt{IsInheritedMethod}(LineConnectionFigure, setAttribute)\\
&\wedge\ \texttt{IsInheritedMethod}(LineConnectionFigure, findFigureInside)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, EllipseFigure, basicTransform)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, EllipseFigure, contains)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, EllipseFigure, setAttribute)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, EllipseFigure, findFigureInside)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, EllipseFigure, addNotify)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, DiamondFigure, basicTransform)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, DiamondFigure, contains)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, DiamondFigure, setAttribute)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, DiamondFigure, findFigureInside)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, DiamondFigure, addNotify)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, RectangleFigure, basicTransform)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, RectangleFigure, contains)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, RectangleFigure, setAttribute)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, RectangleFigure, findFigureInside)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, RectangleFigure, addNotify)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, RoundRectangleFigure, basicTransform)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, RoundRectangleFigure, contains)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, RoundRectangleFigure, setAttribute)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, RoundRectangleFigure, findFigureInside)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, RoundRectangleFigure, addNotify)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, TriangleFigure, basicTransform)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, TriangleFigure, contains)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, TriangleFigure, setAttribute)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, TriangleFigure, findFigureInside)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, TriangleFigure, addNotify)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, TextFigure, basicTransform)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, TextFigure, contains)\\
&\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(RemoveNotifyVisitor, TextFigure, setAttribute)
\end{aligned}
$$

$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, TextFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, TextFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, BezierFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, BezierFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, BezierFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, BezierFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, BezierFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, TextAreaFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, TextAreaFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, TextAreaFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, TextAreaFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, TextAreaFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, NodeFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, NodeFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, NodeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, NodeFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, NodeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, SVGImage, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, SVGImage, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, SVGImage, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, SVGImage, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, SVGImage, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, SVGPath, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, SVGPath, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, SVGPath, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, SVGPath, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, SVGPath, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, DependencyFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, DependencyFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, DependencyFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, DependencyFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, DependencyFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, LineConnectionFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, LineConnectionFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, LineConnectionFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, LineConnectionFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, LineConnectionFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, LabeledLineConnectionFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, LabeledLineConnectionFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, LabeledLineConnectionFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, LabeledLineConnectionFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, LabeledLineConnectionFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, AbstractCompositeFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, AbstractCompositeFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, AbstractCompositeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, AbstractCompositeFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, AbstractCompositeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, GraphicalCompositeFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, GraphicalCompositeFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, GraphicalCompositeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, GraphicalCompositeFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, GraphicalCompositeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, AbstractFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, AbstractFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, AbstractFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, AbstractFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, AbstractFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($RemoveNotifyVisitor, AbstractFigure, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, EllipseFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, EllipseFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, EllipseFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, EllipseFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, EllipseFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, EllipseFigure, addNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, DiamondFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, DiamondFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, DiamondFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, DiamondFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, DiamondFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, DiamondFigure, addNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, RectangleFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, RectangleFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, RectangleFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, RectangleFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, RectangleFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, RectangleFigure, addNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, RoundRectangleFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, RoundRectangleFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, RoundRectangleFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, RoundRectangleFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, RoundRectangleFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, RoundRectangleFigure, addNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TriangleFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TriangleFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TriangleFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TriangleFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TriangleFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TriangleFigure, addNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TextFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TextFigure, basicTransform$))

$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TextFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TextFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TextFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TextFigure, addNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, BezierFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, BezierFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, BezierFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, BezierFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, BezierFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, BezierFigure, addNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TextAreaFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TextAreaFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TextAreaFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TextAreaFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TextAreaFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, TextAreaFigure, addNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, NodeFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, NodeFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, NodeFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, NodeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, NodeFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, NodeFigure, addNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, SVGImage, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, SVGImage, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, SVGImage, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, SVGImage, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, SVGImage, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, SVGImage, addNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, SVGPath, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, SVGPath, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, SVGPath, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, SVGPath, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, SVGPath, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, SVGPath, addNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, DependencyFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, DependencyFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, DependencyFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, DependencyFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, DependencyFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, DependencyFigure, addNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, LineConnectionFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, LineConnectionFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, LineConnectionFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, LineConnectionFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, LineConnectionFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, LineConnectionFigure, addNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, LabeledLineConnectionFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, LabeledLineConnectionFigure, basicTransform$))
$\wedge \neg($IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, LabeledLineConnectionFigure, containsTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, LabeledLineConnectionFigure, contains$))
$\wedge \neg($IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, LabeledLineConnectionFigure, setAttributeTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, LabeledLineConnectionFigure, setAttribute$))
$\wedge \neg($IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, LabeledLineConnectionFigure, findFigureInsideTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, LabeledLineConnectionFigure, findFigureInside$))
$\wedge \neg($IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, LabeledLineConnectionFigure, addNotifyTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, LabeledLineConnectionFigure, addNotify$))
$\wedge \neg($IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, AbstractCompositeFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, AbstractCompositeFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, AbstractCompositeFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, AbstractCompositeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, AbstractCompositeFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, AbstractCompositeFigure, addNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, GraphicalCompositeFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, GraphicalCompositeFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, GraphicalCompositeFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, GraphicalCompositeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, GraphicalCompositeFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($RemoveNotifyVisitor, GraphicalCompositeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, EllipseFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, EllipseFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, EllipseFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, EllipseFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, EllipseFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, DiamondFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, DiamondFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, DiamondFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, DiamondFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, DiamondFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, RectangleFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, RectangleFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, RectangleFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, RectangleFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, RectangleFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, RoundRectangleFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, RoundRectangleFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, RoundRectangleFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, RoundRectangleFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, RoundRectangleFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, TriangleFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($AddNotifyVisitor, TriangleFigure, contains$)

$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, TriangleFigure, setAttribute)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, TriangleFigure, findFigureInside)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, TriangleFigure, removeNotify)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, TextFigure, basicTransform)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, TextFigure, contains)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, TextFigure, setAttribute)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, TextFigure, findFigureInside)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, TextFigure, removeNotify)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, BezierFigure, basicTransform)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, BezierFigure, contains)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, BezierFigure, setAttribute)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, BezierFigure, findFigureInside)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, BezierFigure, removeNotify)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, TextAreaFigure, basicTransform)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, TextAreaFigure, contains)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, TextAreaFigure, setAttribute)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, TextAreaFigure, findFigureInside)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, TextAreaFigure, removeNotify)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, NodeFigure, basicTransform)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, NodeFigure, contains)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, NodeFigure, setAttribute)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, NodeFigure, findFigureInside)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, NodeFigure, removeNotify)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, SVGImage, basicTransform)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, SVGImage, contains)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, SVGImage, setAttribute)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, SVGImage, findFigureInside)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, SVGImage, removeNotify)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, SVGPath, basicTransform)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, SVGPath, contains)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, SVGPath, setAttribute)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, SVGPath, findFigureInside)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, SVGPath, removeNotify)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, DependencyFigure, basicTransform)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, DependencyFigure, contains)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, DependencyFigure, setAttribute)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, DependencyFigure, findFigureInside)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, DependencyFigure, removeNotify)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, LineConnectionFigure, basicTransform)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, LineConnectionFigure, contains)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, LineConnectionFigure, setAttribute)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, LineConnectionFigure, findFigureInside)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, LineConnectionFigure, removeNotify)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, LabeledLineConnectionFigure, basicTransform)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, LabeledLineConnectionFigure, contains)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, LabeledLineConnectionFigure, setAttribute)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, LabeledLineConnectionFigure, findFigureInside)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, LabeledLineConnectionFigure, removeNotify)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, AbstractCompositeFigure, basicTransform)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, AbstractCompositeFigure, contains)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, AbstractCompositeFigure, setAttribute)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, AbstractCompositeFigure, findFigureInside)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, AbstractCompositeFigure, removeNotify)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, GraphicalCompositeFigure, basicTransform)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, GraphicalCompositeFigure, contains)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, GraphicalCompositeFigure, setAttribute)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, GraphicalCompositeFigure, findFigureInside)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, GraphicalCompositeFigure, removeNotify)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, AbstractFigure, basicTransform)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, AbstractFigure, contains)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, AbstractFigure, setAttribute)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, AbstractFigure, findFigureInside)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, AbstractFigure, addNotify)$
$\wedge\ \neg\texttt{IsUsedConstructorAsMethodParameter}(AddNotifyVisitor, AbstractFigure, removeNotify)$
$\wedge\ \neg(\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, EllipseFigure, basicTransformTmpVC)$
$\quad\vee\ \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, EllipseFigure, basicTransform))$
$\wedge\ \neg\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, EllipseFigure, contains)$
$\wedge\ \neg\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, EllipseFigure, setAttribute)$
$\wedge\ \neg\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, EllipseFigure, findFigureInside)$
$\wedge\ \neg\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, EllipseFigure, removeNotify)$
$\wedge\ \neg(\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, DiamondFigure, basicTransformTmpVC)$
$\quad\vee\ \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, DiamondFigure, basicTransform))$
$\wedge\ \neg\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, DiamondFigure, contains)$
$\wedge\ \neg\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, DiamondFigure, setAttribute)$
$\wedge\ \neg\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, DiamondFigure, findFigureInside)$
$\wedge\ \neg\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, DiamondFigure, removeNotify)$
$\wedge\ \neg(\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, RectangleFigure, basicTransformTmpVC)$
$\quad\vee\ \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, RectangleFigure, basicTransform))$
$\wedge\ \neg\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, RectangleFigure, contains)$
$\wedge\ \neg\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, RectangleFigure, setAttribute)$
$\wedge\ \neg\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, RectangleFigure, findFigureInside)$
$\wedge\ \neg\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, RectangleFigure, removeNotify)$
$\wedge\ \neg(\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, RoundRectangleFigure, basicTransformTmpVC)$
$\quad\vee\ \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, RoundRectangleFigure, basicTransform))$
$\wedge\ \neg\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, RoundRectangleFigure, contains)$
$\wedge\ \neg\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, RoundRectangleFigure, setAttribute)$
$\wedge\ \neg\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, RoundRectangleFigure, findFigureInside)$
$\wedge\ \neg\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, RoundRectangleFigure, removeNotify)$
$\wedge\ \neg(\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TriangleFigure, basicTransformTmpVC)$
$\quad\vee\ \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TriangleFigure, basicTransform))$

$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TriangleFigure, contains)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TriangleFigure, setAttribute)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TriangleFigure, findFigureInside)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TriangleFigure, removeNotify)$
$\wedge \neg (\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TextFigure, basicTransformTmpVC)$
$\quad \vee \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TextFigure, basicTransform))$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TextFigure, contains)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TextFigure, setAttribute)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TextFigure, findFigureInside)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TextFigure, removeNotify)$
$\wedge \neg (\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, BezierFigure, basicTransformTmpVC)$
$\quad \vee \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, BezierFigure, basicTransform))$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, BezierFigure, contains)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, BezierFigure, setAttribute)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, BezierFigure, findFigureInside)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, BezierFigure, removeNotify)$
$\wedge \neg (\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TextAreaFigure, basicTransformTmpVC)$
$\quad \vee \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TextAreaFigure, basicTransform))$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TextAreaFigure, contains)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TextAreaFigure, setAttribute)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TextAreaFigure, findFigureInside)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, TextAreaFigure, removeNotify)$
$\wedge \neg (\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, NodeFigure, basicTransformTmpVC)$
$\quad \vee \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, NodeFigure, basicTransform))$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, NodeFigure, contains)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, NodeFigure, setAttribute)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, NodeFigure, findFigureInside)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, NodeFigure, removeNotify)$
$\wedge \neg (\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, SVGImage, basicTransformTmpVC)$
$\quad \vee \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, SVGImage, basicTransform))$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, SVGImage, contains)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, SVGImage, setAttribute)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, SVGImage, findFigureInside)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, SVGImage, removeNotify)$
$\wedge \neg (\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, SVGPath, basicTransformTmpVC)$
$\quad \vee \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, SVGPath, basicTransform))$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, SVGPath, contains)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, SVGPath, setAttribute)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, SVGPath, findFigureInside)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, SVGPath, removeNotify)$
$\wedge \neg (\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, DependencyFigure, basicTransformTmpVC)$
$\quad \vee \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, DependencyFigure, basicTransform))$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, DependencyFigure, contains)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, DependencyFigure, setAttribute)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, DependencyFigure, findFigureInside)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, DependencyFigure, removeNotify)$
$\wedge \neg (\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, LineConnectionFigure, basicTransformTmpVC)$
$\quad \vee \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, LineConnectionFigure, basicTransform))$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, LineConnectionFigure, contains)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, LineConnectionFigure, setAttribute)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, LineConnectionFigure, findFigureInside)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, LineConnectionFigure, removeNotify)$
$\wedge \neg (\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, LabeledLineConnectionFigure, basicTransformTmpVC)$
$\quad \vee \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, LabeledLineConnectionFigure, basicTransform))$
$\wedge \neg (\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, LabeledLineConnectionFigure, containsTmpVC)$
$\quad \vee \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, LabeledLineConnectionFigure, contains))$
$\wedge \neg (\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, LabeledLineConnectionFigure, setAttributeTmpVC)$
$\quad \vee \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, LabeledLineConnectionFigure, setAttribute))$
$\wedge \neg (\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, LabeledLineConnectionFigure, findFigureInsideTmpVC)$
$\quad \vee \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, LabeledLineConnectionFigure, findFigureInside))$
$\wedge \neg (\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, LabeledLineConnectionFigure, removeNotifyTmpVC)$
$\quad \vee \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, LabeledLineConnectionFigure, removeNotify))$
$\wedge \neg (\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, AbstractCompositeFigure, basicTransformTmpVC)$
$\quad \vee \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, AbstractCompositeFigure, basicTransform))$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, AbstractCompositeFigure, contains)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, AbstractCompositeFigure, setAttribute)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, AbstractCompositeFigure, findFigureInside)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, AbstractCompositeFigure, removeNotify)$
$\wedge \neg (\texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, GraphicalCompositeFigure, basicTransformTmpVC)$
$\quad \vee \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, GraphicalCompositeFigure, basicTransform))$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, GraphicalCompositeFigure, contains)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, GraphicalCompositeFigure, setAttribute)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, GraphicalCompositeFigure, findFigureInside)$
$\wedge \neg \texttt{IsUsedConstructorAsObjectReceiver}(AddNotifyVisitor, GraphicalCompositeFigure, removeNotify)$
$\wedge \neg \texttt{IsUsedConstructorAsMethodParameter}(FindFigureInsideVisitor, EllipseFigure, basicTransform)$
$\wedge \neg \texttt{IsUsedConstructorAsMethodParameter}(FindFigureInsideVisitor, EllipseFigure, contains)$
$\wedge \neg \texttt{IsUsedConstructorAsMethodParameter}(FindFigureInsideVisitor, EllipseFigure, setAttribute)$
$\wedge \neg \texttt{IsUsedConstructorAsMethodParameter}(FindFigureInsideVisitor, EllipseFigure, addNotify)$
$\wedge \neg \texttt{IsUsedConstructorAsMethodParameter}(FindFigureInsideVisitor, EllipseFigure, removeNotify)$
$\wedge \neg \texttt{IsUsedConstructorAsMethodParameter}(FindFigureInsideVisitor, DiamondFigure, basicTransform)$
$\wedge \neg \texttt{IsUsedConstructorAsMethodParameter}(FindFigureInsideVisitor, DiamondFigure, contains)$
$\wedge \neg \texttt{IsUsedConstructorAsMethodParameter}(FindFigureInsideVisitor, DiamondFigure, setAttribute)$
$\wedge \neg \texttt{IsUsedConstructorAsMethodParameter}(FindFigureInsideVisitor, DiamondFigure, addNotify)$
$\wedge \neg \texttt{IsUsedConstructorAsMethodParameter}(FindFigureInsideVisitor, DiamondFigure, removeNotify)$
$\wedge \neg \texttt{IsUsedConstructorAsMethodParameter}(FindFigureInsideVisitor, RectangleFigure, basicTransform)$
$\wedge \neg \texttt{IsUsedConstructorAsMethodParameter}(FindFigureInsideVisitor, RectangleFigure, contains)$
$\wedge \neg \texttt{IsUsedConstructorAsMethodParameter}(FindFigureInsideVisitor, RectangleFigure, setAttribute)$
$\wedge \neg \texttt{IsUsedConstructorAsMethodParameter}(FindFigureInsideVisitor, RectangleFigure, addNotify)$
$\wedge \neg \texttt{IsUsedConstructorAsMethodParameter}(FindFigureInsideVisitor, RectangleFigure, removeNotify)$
$\wedge \neg \texttt{IsUsedConstructorAsMethodParameter}(FindFigureInsideVisitor, RoundRectangleFigure, basicTransform)$

$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, RoundRectangleFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, RoundRectangleFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, RoundRectangleFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, RoundRectangleFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, TriangleFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, TriangleFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, TriangleFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, TriangleFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, TriangleFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, TextFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, TextFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, TextFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, TextFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, TextFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, BezierFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, BezierFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, BezierFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, BezierFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, BezierFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, TextAreaFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, TextAreaFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, TextAreaFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, TextAreaFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, TextAreaFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, NodeFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, NodeFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, NodeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, NodeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, NodeFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, SVGImage, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, SVGImage, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, SVGImage, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, SVGImage, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, SVGImage, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, SVGPath, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, SVGPath, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, SVGPath, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, SVGPath, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, SVGPath, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, DependencyFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, DependencyFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, DependencyFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, DependencyFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, DependencyFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, LineConnectionFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, LineConnectionFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, LineConnectionFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, LineConnectionFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, LineConnectionFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, LabeledLineConnectionFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, LabeledLineConnectionFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, LabeledLineConnectionFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, LabeledLineConnectionFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, LabeledLineConnectionFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, AbstractCompositeFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, AbstractCompositeFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, AbstractCompositeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, AbstractCompositeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, AbstractCompositeFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, GraphicalCompositeFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, GraphicalCompositeFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, GraphicalCompositeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, GraphicalCompositeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, GraphicalCompositeFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, AbstractFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, AbstractFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, AbstractFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, AbstractFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, AbstractFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($FindFigureInsideVisitor, AbstractFigure, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, EllipseFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, EllipseFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, EllipseFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, EllipseFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, EllipseFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, EllipseFigure, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, DiamondFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, DiamondFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, DiamondFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, DiamondFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, DiamondFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, DiamondFigure, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, RectangleFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, RectangleFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, RectangleFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, RectangleFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, RectangleFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, RectangleFigure, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, RoundRectangleFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, RoundRectangleFigure, basicTransform$))

$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, RoundRectangleFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, RoundRectangleFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, RoundRectangleFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, RoundRectangleFigure, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TriangleFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TriangleFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TriangleFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TriangleFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TriangleFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TriangleFigure, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TextFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TextFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TextFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TextFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TextFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TextFigure, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, BezierFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, BezierFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, BezierFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, BezierFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, BezierFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, BezierFigure, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TextAreaFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TextAreaFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TextAreaFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TextAreaFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TextAreaFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, TextAreaFigure, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, NodeFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, NodeFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, NodeFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, NodeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, NodeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, NodeFigure, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, SVGImage, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, SVGImage, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, SVGImage, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, SVGImage, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, SVGImage, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, SVGImage, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, SVGPath, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, SVGPath, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, SVGPath, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, SVGPath, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, SVGPath, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, SVGPath, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, DependencyFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, DependencyFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, DependencyFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, DependencyFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, DependencyFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, DependencyFigure, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, LineConnectionFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, LineConnectionFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, LineConnectionFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, LineConnectionFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, LineConnectionFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, LineConnectionFigure, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, LabeledLineConnectionFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, LabeledLineConnectionFigure, basicTransform$))
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, LabeledLineConnectionFigure, containsTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, LabeledLineConnectionFigure, contains$))
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, LabeledLineConnectionFigure, setAttributeTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, LabeledLineConnectionFigure, setAttribute$))
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, LabeledLineConnectionFigure, addNotifyTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, LabeledLineConnectionFigure, addNotify$))
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, LabeledLineConnectionFigure, removeNotifyTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, LabeledLineConnectionFigure, removeNotify$))
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, AbstractCompositeFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, AbstractCompositeFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, AbstractCompositeFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, AbstractCompositeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, AbstractCompositeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, AbstractCompositeFigure, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, GraphicalCompositeFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, GraphicalCompositeFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, GraphicalCompositeFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, GraphicalCompositeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, GraphicalCompositeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($FindFigureInsideVisitor, GraphicalCompositeFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, EllipseFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, EllipseFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, EllipseFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, EllipseFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, EllipseFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, DiamondFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, DiamondFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, DiamondFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, DiamondFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, DiamondFigure, removeNotify$)

$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, RectangleFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, RectangleFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, RectangleFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, RectangleFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, RectangleFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, RoundRectangleFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, RoundRectangleFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, RoundRectangleFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, RoundRectangleFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, RoundRectangleFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, TriangleFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, TriangleFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, TriangleFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, TriangleFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, TriangleFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, TextFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, TextFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, TextFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, TextFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, TextFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, BezierFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, BezierFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, BezierFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, BezierFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, BezierFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, TextAreaFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, TextAreaFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, TextAreaFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, TextAreaFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, TextAreaFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, NodeFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, NodeFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, NodeFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, NodeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, NodeFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, SVGImage, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, SVGImage, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, SVGImage, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, SVGImage, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, SVGImage, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, SVGPath, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, SVGPath, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, SVGPath, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, SVGPath, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, SVGPath, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, DependencyFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, DependencyFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, DependencyFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, DependencyFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, DependencyFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, LineConnectionFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, LineConnectionFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, LineConnectionFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, LineConnectionFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, LineConnectionFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, LabeledLineConnectionFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, LabeledLineConnectionFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, LabeledLineConnectionFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, LabeledLineConnectionFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, LabeledLineConnectionFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, AbstractCompositeFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, AbstractCompositeFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, AbstractCompositeFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, AbstractCompositeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, AbstractCompositeFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, GraphicalCompositeFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, GraphicalCompositeFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, GraphicalCompositeFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, GraphicalCompositeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, GraphicalCompositeFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, AbstractFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, AbstractFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, AbstractFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, AbstractFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, AbstractFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($SetAttributeVisitor, AbstractFigure, removeNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($SetAttributeVisitor, EllipseFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor, EllipseFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor, EllipseFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor, EllipseFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor, EllipseFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor, EllipseFigure, removeNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($SetAttributeVisitor, DiamondFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor, DiamondFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor, DiamondFigure, contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor, DiamondFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor, DiamondFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor, DiamondFigure, removeNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($SetAttributeVisitor, RectangleFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor, RectangleFigure, basicTransform$))

$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $RectangleFigure$, $contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $RectangleFigure$, $findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $RectangleFigure$, $addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $RectangleFigure$, $removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $RoundRectangleFigure$, $basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $RoundRectangleFigure$, $basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $RoundRectangleFigure$, $contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $RoundRectangleFigure$, $findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $RoundRectangleFigure$, $addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $RoundRectangleFigure$, $removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TriangleFigure$, $basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TriangleFigure$, $basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TriangleFigure$, $contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TriangleFigure$, $findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TriangleFigure$, $addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TriangleFigure$, $removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TextFigure$, $basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TextFigure$, $basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TextFigure$, $contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TextFigure$, $findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TextFigure$, $addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TextFigure$, $removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $BezierFigure$, $basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $BezierFigure$, $basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $BezierFigure$, $contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $BezierFigure$, $findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $BezierFigure$, $addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $BezierFigure$, $removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TextAreaFigure$, $basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TextAreaFigure$, $basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TextAreaFigure$, $contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TextAreaFigure$, $findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TextAreaFigure$, $addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $TextAreaFigure$, $removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $NodeFigure$, $basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $NodeFigure$, $basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $NodeFigure$, $contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $NodeFigure$, $findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $NodeFigure$, $addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $NodeFigure$, $removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $SVGImage$, $basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $SVGImage$, $basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $SVGImage$, $contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $SVGImage$, $findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $SVGImage$, $addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $SVGImage$, $removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $SVGPath$, $basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $SVGPath$, $basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $SVGPath$, $contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $SVGPath$, $findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $SVGPath$, $addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $SVGPath$, $removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $DependencyFigure$, $basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $DependencyFigure$, $basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $DependencyFigure$, $contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $DependencyFigure$, $findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $DependencyFigure$, $addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $DependencyFigure$, $removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $LineConnectionFigure$, $basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $LineConnectionFigure$, $basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $LineConnectionFigure$, $contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $LineConnectionFigure$, $findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $LineConnectionFigure$, $addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $LineConnectionFigure$, $removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $LabeledLineConnectionFigure$, $basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $LabeledLineConnectionFigure$, $basicTransform$))
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $LabeledLineConnectionFigure$, $containsTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $LabeledLineConnectionFigure$, $contains$))
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $LabeledLineConnectionFigure$, $findFigureInsideTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $LabeledLineConnectionFigure$, $findFigureInside$))
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $LabeledLineConnectionFigure$, $addNotifyTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $LabeledLineConnectionFigure$, $addNotify$))
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $LabeledLineConnectionFigure$, $removeNotifyTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $LabeledLineConnectionFigure$, $removeNotify$))
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $AbstractCompositeFigure$, $basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $AbstractCompositeFigure$, $basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $AbstractCompositeFigure$, $contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $AbstractCompositeFigure$, $findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $AbstractCompositeFigure$, $addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $AbstractCompositeFigure$, $removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $GraphicalCompositeFigure$, $basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $GraphicalCompositeFigure$, $basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $GraphicalCompositeFigure$, $contains$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $GraphicalCompositeFigure$, $findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $GraphicalCompositeFigure$, $addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($SetAttributeVisitor$, $GraphicalCompositeFigure$, $removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor$, $EllipseFigure$, $basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor$, $EllipseFigure$, $setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor$, $EllipseFigure$, $findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor$, $EllipseFigure$, $addNotify$)

$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, EllipseFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, DiamondFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, DiamondFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, DiamondFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, DiamondFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, DiamondFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, RectangleFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, RectangleFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, RectangleFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, RectangleFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, RectangleFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, RoundRectangleFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, RoundRectangleFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, RoundRectangleFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, RoundRectangleFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, RoundRectangleFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, TriangleFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, TriangleFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, TriangleFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, TriangleFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, TriangleFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, TextFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, TextFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, TextFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, TextFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, TextFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, BezierFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, BezierFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, BezierFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, BezierFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, BezierFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, TextAreaFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, TextAreaFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, TextAreaFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, TextAreaFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, TextAreaFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, NodeFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, NodeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, NodeFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, NodeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, NodeFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, SVGImage, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, SVGImage, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, SVGImage, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, SVGImage, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, SVGImage, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, SVGPath, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, SVGPath, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, SVGPath, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, SVGPath, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, SVGPath, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, DependencyFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, DependencyFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, DependencyFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, DependencyFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, DependencyFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, LineConnectionFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, LineConnectionFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, LineConnectionFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, LineConnectionFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, LineConnectionFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, LabeledLineConnectionFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, LabeledLineConnectionFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, LabeledLineConnectionFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, LabeledLineConnectionFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, LabeledLineConnectionFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, AbstractCompositeFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, AbstractCompositeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, AbstractCompositeFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, AbstractCompositeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, AbstractCompositeFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, GraphicalCompositeFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, GraphicalCompositeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, GraphicalCompositeFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, GraphicalCompositeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, GraphicalCompositeFigure, removeNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, AbstractFigure, basicTransform$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, AbstractFigure, contains$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, AbstractFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, AbstractFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, AbstractFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsMethodParameter($ContainsVisitor, AbstractFigure, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($ContainsVisitor, EllipseFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, EllipseFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, EllipseFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, EllipseFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, EllipseFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, EllipseFigure, removeNotify$)
$\wedge \neg$(IsUsedConstructorAsObjectReceiver($ContainsVisitor, DiamondFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, DiamondFigure, basicTransform$))

$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, DiamondFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, DiamondFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, DiamondFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, DiamondFigure, removeNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, RectangleFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, RectangleFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, RectangleFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, RectangleFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, RectangleFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, RectangleFigure, removeNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, RoundRectangleFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, RoundRectangleFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, RoundRectangleFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, RoundRectangleFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, RoundRectangleFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, RoundRectangleFigure, removeNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, TriangleFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, TriangleFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, TriangleFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, TriangleFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, TriangleFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, TriangleFigure, removeNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, TextFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, TextFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, TextFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, TextFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, TextFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, TextFigure, removeNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, BezierFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, BezierFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, BezierFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, BezierFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, BezierFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, BezierFigure, removeNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, TextAreaFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, TextAreaFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, TextAreaFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, TextAreaFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, TextAreaFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, TextAreaFigure, removeNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, NodeFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, NodeFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, NodeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, NodeFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, NodeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, NodeFigure, removeNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, SVGImage, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, SVGImage, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, SVGImage, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, SVGImage, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, SVGImage, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, SVGImage, removeNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, SVGPath, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, SVGPath, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, SVGPath, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, SVGPath, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, SVGPath, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, SVGPath, removeNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, DependencyFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, DependencyFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, DependencyFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, DependencyFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, DependencyFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, DependencyFigure, removeNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, LineConnectionFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, LineConnectionFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, LineConnectionFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, LineConnectionFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, LineConnectionFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, LineConnectionFigure, removeNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, LabeledLineConnectionFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, LabeledLineConnectionFigure, basicTransform$))
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, LabeledLineConnectionFigure, setAttributeTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, LabeledLineConnectionFigure, setAttribute$))
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, LabeledLineConnectionFigure, findFigureInsideTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, LabeledLineConnectionFigure, findFigureInside$))
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, LabeledLineConnectionFigure, addNotifyTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, LabeledLineConnectionFigure, addNotify$))
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, LabeledLineConnectionFigure, removeNotifyTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, LabeledLineConnectionFigure, removeNotify$))
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, AbstractCompositeFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, AbstractCompositeFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, AbstractCompositeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, AbstractCompositeFigure, findFigureInside$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, AbstractCompositeFigure, addNotify$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, AbstractCompositeFigure, removeNotify$)
$\wedge \neg($IsUsedConstructorAsObjectReceiver($ContainsVisitor, GraphicalCompositeFigure, basicTransformTmpVC$)
$\vee$ IsUsedConstructorAsObjectReceiver($ContainsVisitor, GraphicalCompositeFigure, basicTransform$))
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, GraphicalCompositeFigure, setAttribute$)
$\wedge \neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, GraphicalCompositeFigure, findFigureInside$)

$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, GraphicalCompositeFigure, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($ContainsVisitor, GraphicalCompositeFigure, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, EllipseFigure, contains$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, EllipseFigure, setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, EllipseFigure, findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, EllipseFigure, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, EllipseFigure, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, DiamondFigure, contains$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, DiamondFigure, setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, DiamondFigure, findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, DiamondFigure, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, DiamondFigure, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, RectangleFigure, contains$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, RectangleFigure, setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, RectangleFigure, findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, RectangleFigure, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, RectangleFigure, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, RoundRectangleFigure, contains$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, RoundRectangleFigure, setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, RoundRectangleFigure, findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, RoundRectangleFigure, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, RoundRectangleFigure, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, TriangleFigure, contains$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, TriangleFigure, setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, TriangleFigure, findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, TriangleFigure, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, TriangleFigure, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, TextFigure, contains$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, TextFigure, setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, TextFigure, findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, TextFigure, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, TextFigure, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, BezierFigure, contains$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, BezierFigure, setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, BezierFigure, findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, BezierFigure, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, BezierFigure, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, TextAreaFigure, contains$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, TextAreaFigure, setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, TextAreaFigure, findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, TextAreaFigure, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, TextAreaFigure, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, NodeFigure, contains$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, NodeFigure, setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, NodeFigure, findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, NodeFigure, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, NodeFigure, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, SVGImage, contains$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, SVGImage, setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, SVGImage, findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, SVGImage, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, SVGImage, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, SVGPath, contains$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, SVGPath, setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, SVGPath, findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, SVGPath, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, SVGPath, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, DependencyFigure, contains$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, DependencyFigure, setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, DependencyFigure, findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, DependencyFigure, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, DependencyFigure, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, LineConnectionFigure, contains$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, LineConnectionFigure, setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, LineConnectionFigure, findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, LineConnectionFigure, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, LineConnectionFigure, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, LabeledLineConnectionFigure, contains$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, LabeledLineConnectionFigure, setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, LabeledLineConnectionFigure, findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, LabeledLineConnectionFigure, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, LabeledLineConnectionFigure, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, AbstractCompositeFigure, contains$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, AbstractCompositeFigure, setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, AbstractCompositeFigure, findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, AbstractCompositeFigure, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, AbstractCompositeFigure, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, GraphicalCompositeFigure, contains$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, GraphicalCompositeFigure, setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, GraphicalCompositeFigure, findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, GraphicalCompositeFigure, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, GraphicalCompositeFigure, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, AbstractFigure, basicTransform$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, AbstractFigure, contains$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, AbstractFigure, setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, AbstractFigure, findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, AbstractFigure, addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsMethodParameter($BasicTransformVisitor, AbstractFigure, removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor, EllipseFigure, contains$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor, EllipseFigure, setAttribute$)

$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $EllipseFigure$, $findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $EllipseFigure$, $addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $EllipseFigure$, $removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $DiamondFigure$, $contains$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $DiamondFigure$, $setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $DiamondFigure$, $findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $DiamondFigure$, $addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $DiamondFigure$, $removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $RectangleFigure$, $contains$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $RectangleFigure$, $setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $RectangleFigure$, $findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $RectangleFigure$, $addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $RectangleFigure$, $removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $RoundRectangleFigure$, $contains$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $RoundRectangleFigure$, $setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $RoundRectangleFigure$, $findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $RoundRectangleFigure$, $addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $RoundRectangleFigure$, $removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $TriangleFigure$, $contains$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $TriangleFigure$, $setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $TriangleFigure$, $findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $TriangleFigure$, $addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $TriangleFigure$, $removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $TextFigure$, $contains$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $TextFigure$, $setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $TextFigure$, $findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $TextFigure$, $addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $TextFigure$, $removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $BezierFigure$, $contains$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $BezierFigure$, $setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $BezierFigure$, $findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $BezierFigure$, $addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $BezierFigure$, $removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $TextAreaFigure$, $contains$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $TextAreaFigure$, $setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $TextAreaFigure$, $findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $TextAreaFigure$, $addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $TextAreaFigure$, $removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $NodeFigure$, $contains$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $NodeFigure$, $setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $NodeFigure$, $findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $NodeFigure$, $addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $NodeFigure$, $removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $SVGImage$, $contains$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $SVGImage$, $setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $SVGImage$, $findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $SVGImage$, $addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $SVGImage$, $removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $SVGPath$, $contains$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $SVGPath$, $setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $SVGPath$, $findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $SVGPath$, $addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $SVGPath$, $removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $DependencyFigure$, $contains$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $DependencyFigure$, $setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $DependencyFigure$, $findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $DependencyFigure$, $addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $DependencyFigure$, $removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $LineConnectionFigure$, $contains$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $LineConnectionFigure$, $setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $LineConnectionFigure$, $findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $LineConnectionFigure$, $addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $LineConnectionFigure$, $removeNotify$)
$\wedge$ $\neg$(IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $LabeledLineConnectionFigure$, $containsTmpVC$)
  $\vee$ IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $LabeledLineConnectionFigure$, $contains$))
$\wedge$ $\neg$(IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $LabeledLineConnectionFigure$, $setAttributeTmpVC$)
  $\vee$ IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $LabeledLineConnectionFigure$, $setAttribute$))
$\wedge$ $\neg$(IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $LabeledLineConnectionFigure$, $findFigureInsideTmpVC$)
  $\vee$ IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $LabeledLineConnectionFigure$, $findFigureInside$))
$\wedge$ $\neg$(IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $LabeledLineConnectionFigure$, $addNotifyTmpVC$)
  $\vee$ IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $LabeledLineConnectionFigure$, $addNotify$))
$\wedge$ $\neg$(IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $LabeledLineConnectionFigure$, $removeNotifyTmpVC$)
  $\vee$ IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $LabeledLineConnectionFigure$, $removeNotify$))
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $AbstractCompositeFigure$, $contains$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $AbstractCompositeFigure$, $setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $AbstractCompositeFigure$, $findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $AbstractCompositeFigure$, $addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $AbstractCompositeFigure$, $removeNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $GraphicalCompositeFigure$, $contains$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $GraphicalCompositeFigure$, $setAttribute$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $GraphicalCompositeFigure$, $findFigureInside$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $GraphicalCompositeFigure$, $addNotify$)
$\wedge$ $\neg$IsUsedConstructorAsObjectReceiver($BasicTransformVisitor$, $GraphicalCompositeFigure$, $removeNotify$)
$\wedge$ ExistsType($AbstractCompositeFigure$)
$\wedge$ ExistsType($SVGImage$)
$\wedge$ ExistsType($TextAreaFigure$)
$\wedge$ ExistsType($BezierFigure$)
$\wedge$ ExistsType($TextFigure$)
$\wedge$ ExistsType($TriangleFigure$)
$\wedge$ ExistsType($RoundRectangleFigure$)

$\wedge$ ExistsType($RectangleFigure$)
$\wedge$ ExistsType($DiamondFigure$)
$\wedge$ ExistsType($EllipseFigure$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, removeNotifyTmpVC, LabeledLineConnectionFigure, basicTransform$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, removeNotifyTmpVC, LabeledLineConnectionFigure, contains$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, removeNotifyTmpVC, LabeledLineConnectionFigure, setAttribute$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, removeNotifyTmpVC, LabeledLineConnectionFigure, findFigureInside$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, removeNotifyTmpVC, LabeledLineConnectionFigure, addNotify$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, addNotifyTmpVC, LabeledLineConnectionFigure, basicTransform$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, addNotifyTmpVC, LabeledLineConnectionFigure, contains$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, addNotifyTmpVC, LabeledLineConnectionFigure, setAttribute$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, addNotifyTmpVC, LabeledLineConnectionFigure, findFigureInside$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, addNotifyTmpVC, LabeledLineConnectionFigure, removeNotify$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, findFigureInsideTmpVC, LabeledLineConnectionFigure, basicTransform$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, findFigureInsideTmpVC, LabeledLineConnectionFigure, contains$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, findFigureInsideTmpVC, LabeledLineConnectionFigure, setAttribute$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, findFigureInsideTmpVC, LabeledLineConnectionFigure, addNotify$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, findFigureInsideTmpVC, LabeledLineConnectionFigure, removeNotify$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, setAttributeTmpVC, LabeledLineConnectionFigure, basicTransform$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, setAttributeTmpVC, LabeledLineConnectionFigure, contains$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, setAttributeTmpVC, LabeledLineConnectionFigure, findFigureInside$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, setAttributeTmpVC, LabeledLineConnectionFigure, addNotify$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, setAttributeTmpVC, LabeledLineConnectionFigure, removeNotify$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, containsTmpVC, LabeledLineConnectionFigure, basicTransform$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, containsTmpVC, LabeledLineConnectionFigure, setAttribute$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, containsTmpVC, LabeledLineConnectionFigure, findFigureInside$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, containsTmpVC, LabeledLineConnectionFigure, addNotify$)
$\wedge$ $\neg$ExistsMethodInvocation($LabeledLineConnectionFigure, containsTmpVC, LabeledLineConnectionFigure, removeNotify$)
$\wedge$ $\neg$IsUsedMethod($AbstractFigure, accept\_RemoveNotifyVisitor\_addspecializedMethod\_tmp, [RemoveNotifyVisitor]$)
$\wedge$ $\neg$IsUsedMethod($AbstractFigure, accept\_AddNotifyVisitor\_addspecializedMethod\_tmp, [AddNotifyVisitor]$)
$\wedge$ $\neg$IsUsedMethod($AbstractFigure, accept\_FindFigureInsideVisitor\_addspecializedMethod\_tmp, [FindFigureInsideVisitor]$)
$\wedge$ $\neg$IsUsedMethod($AbstractFigure, accept\_SetAttributeVisitor\_addspecializedMethod\_tmp, [SetAttributeVisitor]$)
$\wedge$ $\neg$IsUsedMethod($AbstractFigure, accept\_ContainsVisitor\_addspecializedMethod\_tmp, [ContainsVisitor]$)
$\wedge$ $\neg$IsUsedMethod($AbstractFigure, accept\_BasicTransformVisitor\_addspecializedMethod\_tmp, [BasicTransformVisitor]$)
$\wedge$AllSubclasses($AbstractFigure$, [$EllipseFigure$; $DiamondFigure$; $RectangleFigure$; $RoundRectangleFigure$; $TriangleFigure$; $TextFigure$; $BezierFigure$;
$\wedge$ $\neg$ExistsMethodDefinition($AbstractFigure, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($LabeledLineConnectionFigure, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($AbstractCompositeFigure, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($GraphicalCompositeFigure, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($EllipseFigure, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($DiamondFigure, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($RectangleFigure, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($RoundRectangleFigure, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($TriangleFigure, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($TextFigure, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($BezierFigure, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($TextAreaFigure, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($NodeFigure, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($SVGImage, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($SVGPath, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($DependencyFigure, accept$)
$\wedge$ $\neg$ExistsMethodDefinition($LineConnectionFigure, accept$)
$\wedge$ $\neg$IsInheritedMethod($AbstractFigure, accept$)
$\wedge$ $\neg$ExistsMethodInvocation($AbstractFigure, removeNotifyTmpVC, AbstractFigure, contains$)
$\wedge$ $\neg$ExistsMethodInvocation($AbstractFigure, removeNotifyTmpVC, AbstractFigure, setAttribute$)
$\wedge$ $\neg$ExistsMethodInvocation($AbstractFigure, removeNotifyTmpVC, AbstractFigure, findFigureInside$)
$\wedge$ $\neg$ExistsMethodInvocation($AbstractFigure, removeNotifyTmpVC, AbstractFigure, addNotify$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, removeNotifyTmpVC, LabeledLineConnectionFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, removeNotifyTmpVC, AbstractCompositeFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, removeNotifyTmpVC, GraphicalCompositeFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, removeNotifyTmpVC, EllipseFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, removeNotifyTmpVC, DiamondFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, removeNotifyTmpVC, RectangleFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, removeNotifyTmpVC, RoundRectangleFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, removeNotifyTmpVC, TriangleFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, removeNotifyTmpVC, TextFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, removeNotifyTmpVC, BezierFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, removeNotifyTmpVC, TextAreaFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, removeNotifyTmpVC, NodeFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, removeNotifyTmpVC, SVGImage$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, removeNotifyTmpVC, SVGPath$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, removeNotifyTmpVC, DependencyFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, removeNotifyTmpVC, LineConnectionFigure$)
$\wedge$ $\neg$ExistsMethodInvocation($AbstractFigure, addNotifyTmpVC, AbstractFigure, contains$)
$\wedge$ $\neg$ExistsMethodInvocation($AbstractFigure, addNotifyTmpVC, AbstractFigure, setAttribute$)
$\wedge$ $\neg$ExistsMethodInvocation($AbstractFigure, addNotifyTmpVC, AbstractFigure, findFigureInside$)
$\wedge$ $\neg$ExistsMethodInvocation($AbstractFigure, addNotifyTmpVC, AbstractFigure, removeNotify$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, addNotifyTmpVC, LabeledLineConnectionFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, addNotifyTmpVC, AbstractCompositeFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, addNotifyTmpVC, GraphicalCompositeFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, addNotifyTmpVC, EllipseFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, addNotifyTmpVC, DiamondFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, addNotifyTmpVC, RectangleFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, addNotifyTmpVC, RoundRectangleFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, addNotifyTmpVC, TriangleFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, addNotifyTmpVC, TextFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, addNotifyTmpVC, BezierFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, addNotifyTmpVC, TextAreaFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, addNotifyTmpVC, NodeFigure$)
$\wedge$ $\neg$IsUsedMethodIn($AbstractFigure, addNotifyTmpVC, SVGImage$)

$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, addNotifyTmpVC, SVGPath$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, addNotifyTmpVC, DependencyFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, addNotifyTmpVC, LineConnectionFigure$)
$\wedge\ \neg$ExistsMethodInvocation($AbstractFigure, findFigureInsideTmpVC, AbstractFigure, contains$)
$\wedge\ \neg$ExistsMethodInvocation($AbstractFigure, findFigureInsideTmpVC, AbstractFigure, setAttribute$)
$\wedge\ \neg$ExistsMethodInvocation($AbstractFigure, findFigureInsideTmpVC, AbstractFigure, addNotify$)
$\wedge\ \neg$ExistsMethodInvocation($AbstractFigure, findFigureInsideTmpVC, AbstractFigure, removeNotify$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, findFigureInsideTmpVC, LabeledLineConnectionFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, findFigureInsideTmpVC, AbstractCompositeFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, findFigureInsideTmpVC, GraphicalCompositeFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, findFigureInsideTmpVC, EllipseFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, findFigureInsideTmpVC, DiamondFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, findFigureInsideTmpVC, RectangleFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, findFigureInsideTmpVC, RoundRectangleFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, findFigureInsideTmpVC, TriangleFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, findFigureInsideTmpVC, TextFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, findFigureInsideTmpVC, BezierFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, findFigureInsideTmpVC, TextAreaFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, findFigureInsideTmpVC, NodeFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, findFigureInsideTmpVC, SVGImage$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, findFigureInsideTmpVC, SVGPath$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, findFigureInsideTmpVC, DependencyFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, findFigureInsideTmpVC, LineConnectionFigure$)
$\wedge\ \neg$ExistsMethodInvocation($AbstractFigure, setAttributeTmpVC, AbstractFigure, contains$)
$\wedge\ \neg$ExistsMethodInvocation($AbstractFigure, setAttributeTmpVC, AbstractFigure, findFigureInside$)
$\wedge\ \neg$ExistsMethodInvocation($AbstractFigure, setAttributeTmpVC, AbstractFigure, addNotify$)
$\wedge\ \neg$ExistsMethodInvocation($AbstractFigure, setAttributeTmpVC, AbstractFigure, removeNotify$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, setAttributeTmpVC, LabeledLineConnectionFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, setAttributeTmpVC, AbstractCompositeFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, setAttributeTmpVC, GraphicalCompositeFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, setAttributeTmpVC, EllipseFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, setAttributeTmpVC, DiamondFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, setAttributeTmpVC, RectangleFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, setAttributeTmpVC, RoundRectangleFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, setAttributeTmpVC, TriangleFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, setAttributeTmpVC, TextFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, setAttributeTmpVC, BezierFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, setAttributeTmpVC, TextAreaFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, setAttributeTmpVC, NodeFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, setAttributeTmpVC, SVGImage$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, setAttributeTmpVC, SVGPath$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, setAttributeTmpVC, DependencyFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, setAttributeTmpVC, LineConnectionFigure$)
$\wedge\ \neg$ExistsMethodInvocation($AbstractFigure, containsTmpVC, AbstractFigure, setAttribute$)
$\wedge\ \neg$ExistsMethodInvocation($AbstractFigure, containsTmpVC, AbstractFigure, findFigureInside$)
$\wedge\ \neg$ExistsMethodInvocation($AbstractFigure, containsTmpVC, AbstractFigure, addNotify$)
$\wedge\ \neg$ExistsMethodInvocation($AbstractFigure, containsTmpVC, AbstractFigure, removeNotify$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, containsTmpVC, LabeledLineConnectionFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, containsTmpVC, AbstractCompositeFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, containsTmpVC, GraphicalCompositeFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, containsTmpVC, EllipseFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, containsTmpVC, DiamondFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, containsTmpVC, RectangleFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, containsTmpVC, RoundRectangleFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, containsTmpVC, TriangleFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, containsTmpVC, TextFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, containsTmpVC, BezierFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, containsTmpVC, TextAreaFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, containsTmpVC, NodeFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, containsTmpVC, SVGImage$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, containsTmpVC, SVGPath$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, containsTmpVC, DependencyFigure$)
$\wedge\ \neg$IsUsedMethodIn($AbstractFigure, containsTmpVC, LineConnectionFigure$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($LineConnectionFigure, removeNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DependencyFigure, removeNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGImage, removeNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($NodeFigure, removeNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextAreaFigure, removeNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($BezierFigure, removeNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextFigure, removeNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TriangleFigure, removeNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RoundRectangleFigure, removeNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RectangleFigure, removeNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DiamondFigure, removeNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($EllipseFigure, removeNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($GraphicalCompositeFigure, removeNotify, this$)
$\wedge$ AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC($LabeledLineConnectionFigure, removeNotify, this, AbstractFigure$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($LabeledLineConnectionFigure, removeNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGImage, addNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextAreaFigure, addNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($BezierFigure, addNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TriangleFigure, addNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RoundRectangleFigure, addNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RectangleFigure, addNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DiamondFigure, addNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($EllipseFigure, addNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($GraphicalCompositeFigure, addNotify, this$)
$\wedge$ AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC($LabeledLineConnectionFigure, addNotify, this, AbstractFigure$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($LabeledLineConnectionFigure, addNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGImage, findFigureInside, this$)

$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextAreaFigure, findFigureInside, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TriangleFigure, findFigureInside, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RoundRectangleFigure, findFigureInside, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RectangleFigure, findFigureInside, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DiamondFigure, findFigureInside, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($EllipseFigure, findFigureInside, this$)
$\wedge$ AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC($LabeledLineConnectionFigure, findFigureInside, this, AbstractFigure$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGPath, setAttribute, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGImage, setAttribute, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextAreaFigure, setAttribute, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TriangleFigure, setAttribute, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RoundRectangleFigure, setAttribute, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RectangleFigure, setAttribute, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DiamondFigure, setAttribute, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($EllipseFigure, setAttribute, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($GraphicalCompositeFigure, setAttribute, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($AbstractCompositeFigure, setAttribute, this$)
$\wedge$ AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC($LabeledLineConnectionFigure, setAttribute, this, AbstractFigure$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGImage, contains, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextAreaFigure, contains, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TriangleFigure, contains, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RoundRectangleFigure, contains, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RectangleFigure, contains, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DiamondFigure, contains, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($EllipseFigure, contains, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($GraphicalCompositeFigure, contains, this$)
$\wedge$ AllInvokedMethodsOnObjectOInBodyOfMAreDeclaredInC($LabeledLineConnectionFigure, contains, this, AbstractFigure$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($AbstractFigure, removeNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($LabeledLineConnectionFigure, removeNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($AbstractCompositeFigure, removeNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($GraphicalCompositeFigure, removeNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($EllipseFigure, removeNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DiamondFigure, removeNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RectangleFigure, removeNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RoundRectangleFigure, removeNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TriangleFigure, removeNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextFigure, removeNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($BezierFigure, removeNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextAreaFigure, removeNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($NodeFigure, removeNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGImage, removeNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGPath, removeNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DependencyFigure, removeNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($LineConnectionFigure, removeNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($AbstractFigure, addNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($LabeledLineConnectionFigure, addNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($AbstractCompositeFigure, addNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($GraphicalCompositeFigure, addNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($EllipseFigure, addNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DiamondFigure, addNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RectangleFigure, addNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RoundRectangleFigure, addNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TriangleFigure, addNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextFigure, addNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($BezierFigure, addNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextAreaFigure, addNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($NodeFigure, addNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGImage, addNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGPath, addNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DependencyFigure, addNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($LineConnectionFigure, addNotify, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($AbstractFigure, findFigureInside, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($LabeledLineConnectionFigure, findFigureInside, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($AbstractCompositeFigure, findFigureInside, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($GraphicalCompositeFigure, findFigureInside, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($EllipseFigure, findFigureInside, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DiamondFigure, findFigureInside, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RectangleFigure, findFigureInside, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RoundRectangleFigure, findFigureInside, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TriangleFigure, findFigureInside, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextFigure, findFigureInside, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($BezierFigure, findFigureInside, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextAreaFigure, findFigureInside, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($NodeFigure, findFigureInside, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGImage, findFigureInside, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGPath, findFigureInside, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DependencyFigure, findFigureInside, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($LineConnectionFigure, findFigureInside, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($AbstractFigure, setAttribute, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($LabeledLineConnectionFigure, setAttribute, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($AbstractCompositeFigure, setAttribute, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($GraphicalCompositeFigure, setAttribute, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($EllipseFigure, setAttribute, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DiamondFigure, setAttribute, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RectangleFigure, setAttribute, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RoundRectangleFigure, setAttribute, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TriangleFigure, setAttribute, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextFigure, setAttribute, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($BezierFigure, setAttribute, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextAreaFigure, setAttribute, v$)

$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($NodeFigure, setAttribute, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGImage, setAttribute, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGPath, setAttribute, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DependencyFigure, setAttribute, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($LineConnectionFigure, setAttribute, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($AbstractFigure, contains, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($LabeledLineConnectionFigure, contains, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($AbstractCompositeFigure, contains, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($GraphicalCompositeFigure, contains, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($EllipseFigure, contains, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DiamondFigure, contains, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RectangleFigure, contains, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RoundRectangleFigure, contains, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TriangleFigure, contains, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextFigure, contains, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($BezierFigure, contains, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextAreaFigure, contains, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($NodeFigure, contains, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGImage, contains, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGPath, contains, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DependencyFigure, contains, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($LineConnectionFigure, contains, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($AbstractFigure, basicTransform, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($LabeledLineConnectionFigure, basicTransform, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($AbstractCompositeFigure, basicTransform, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($GraphicalCompositeFigure, basicTransform, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($EllipseFigure, basicTransform, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DiamondFigure, basicTransform, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RectangleFigure, basicTransform, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($RoundRectangleFigure, basicTransform, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TriangleFigure, basicTransform, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextFigure, basicTransform, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($BezierFigure, basicTransform, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextAreaFigure, basicTransform, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($NodeFigure, basicTransform, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGImage, basicTransform, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($SVGPath, basicTransform, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($DependencyFigure, basicTransform, v$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($LineConnectionFigure, basicTransform, v$)
$\wedge \neg$IsPrivate($RemoveNotifyVisitor, visit$)
$\wedge \neg$HasParameterType($RemoveNotifyVisitor, Void$)
$\wedge \neg$IsPrivate($AddNotifyVisitor, visit$)
$\wedge \neg$HasParameterType($AddNotifyVisitor, Void$)
$\wedge \neg$IsPrimitiveType($Figure$)
$\wedge \neg$IsPrivate($FindFigureInsideVisitor, visit$)
$\wedge \neg$HasParameterType($FindFigureInsideVisitor, Figure$)
$\wedge \neg$IsPrivate($SetAttributeVisitor, visit$)
$\wedge \neg$HasParameterType($SetAttributeVisitor, Void$)
$\wedge \neg$IsPrimitiveType($Boolean$)
$\wedge \neg$IsPrivate($ContainsVisitor, visit$)
$\wedge \neg$HasParameterType($ContainsVisitor, Boolean$)
$\wedge \neg$ExistsAbstractMethod($Visitor, visit$)
$\wedge \neg$IsPrimitiveType($Void$)
$\wedge \neg$IsPrivate($BasicTransformVisitor, visit$)
$\wedge \neg$HasParameterType($BasicTransformVisitor, Void$)
$\wedge \neg$ExistsType($Visitor$)
$\wedge$ ExistsClass($SVGImage$)
$\wedge$ ExistsClass($TextAreaFigure$)
$\wedge$ ExistsClass($BezierFigure$)
$\wedge$ ExistsClass($TextFigure$)
$\wedge$ ExistsClass($TriangleFigure$)
$\wedge$ ExistsClass($RoundRectangleFigure$)
$\wedge$ ExistsClass($RectangleFigure$)
$\wedge$ ExistsClass($DiamondFigure$)
$\wedge$ ExistsClass($EllipseFigure$)
$\wedge$ ExistsClass($AbstractCompositeFigure$)
$\wedge \neg$ExistsType($RemoveNotifyVisitor$)
$\wedge$ BoundVariableInMethodBody($LabeledLineConnectionFigure, removeNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($GraphicalCompositeFigure, removeNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($EllipseFigure, removeNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($DiamondFigure, removeNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($RectangleFigure, removeNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($RoundRectangleFigure, removeNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($TriangleFigure, removeNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($TextFigure, removeNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($BezierFigure, removeNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($TextAreaFigure, removeNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($NodeFigure, removeNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($SVGImage, removeNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($AbstractCompositeFigure, removeNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($DependencyFigure, removeNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($LineConnectionFigure, removeNotify, Drawingd$)
$\wedge \neg$ExistsType($AddNotifyVisitor$)
$\wedge$ BoundVariableInMethodBody($LabeledLineConnectionFigure, addNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($GraphicalCompositeFigure, addNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($EllipseFigure, addNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($DiamondFigure, addNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($RectangleFigure, addNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($RoundRectangleFigure, addNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($TriangleFigure, addNotify, Drawingd$)

$\wedge$ BoundVariableInMethodBody($BezierFigure, addNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($TextAreaFigure, addNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($TextFigure, addNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($SVGImage, addNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($AbstractCompositeFigure, addNotify, Drawingd$)
$\wedge$ BoundVariableInMethodBody($LineConnectionFigure, addNotify, Drawingd$)
$\wedge \neg$ExistsType($FindFigureInsideVisitor$)
$\wedge$ BoundVariableInMethodBody($EllipseFigure, findFigureInside, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($DiamondFigure, findFigureInside, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($RectangleFigure, findFigureInside, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($RoundRectangleFigure, findFigureInside, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($TriangleFigure, findFigureInside, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($TextAreaFigure, findFigureInside, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($TextFigure, findFigureInside, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($SVGImage, findFigureInside, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($AbstractCompositeFigure, findFigureInside, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($BezierFigure, findFigureInside, Point2D.Doublep$)
$\wedge \neg$ExistsType($SetAttributeVisitor$)
$\wedge$ BoundVariableInMethodBody($AbstractCompositeFigure, setAttribute, AttributeKeykey$)
$\wedge$ BoundVariableInMethodBody($AbstractCompositeFigure, setAttribute, Objectvalue$)
$\wedge$ BoundVariableInMethodBody($GraphicalCompositeFigure, setAttribute, AttributeKeykey$)
$\wedge$ BoundVariableInMethodBody($GraphicalCompositeFigure, setAttribute, Objectvalue$)
$\wedge$ BoundVariableInMethodBody($EllipseFigure, setAttribute, AttributeKeykey$)
$\wedge$ BoundVariableInMethodBody($EllipseFigure, setAttribute, Objectvalue$)
$\wedge$ BoundVariableInMethodBody($DiamondFigure, setAttribute, AttributeKeykey$)
$\wedge$ BoundVariableInMethodBody($DiamondFigure, setAttribute, Objectvalue$)
$\wedge$ BoundVariableInMethodBody($RectangleFigure, setAttribute, AttributeKeykey$)
$\wedge$ BoundVariableInMethodBody($RectangleFigure, setAttribute, Objectvalue$)
$\wedge$ BoundVariableInMethodBody($RoundRectangleFigure, setAttribute, AttributeKeykey$)
$\wedge$ BoundVariableInMethodBody($RoundRectangleFigure, setAttribute, Objectvalue$)
$\wedge$ BoundVariableInMethodBody($TriangleFigure, setAttribute, AttributeKeykey$)
$\wedge$ BoundVariableInMethodBody($TriangleFigure, setAttribute, Objectvalue$)
$\wedge$ BoundVariableInMethodBody($TextAreaFigure, setAttribute, AttributeKeykey$)
$\wedge$ BoundVariableInMethodBody($TextAreaFigure, setAttribute, Objectvalue$)
$\wedge$ BoundVariableInMethodBody($TextFigure, setAttribute, AttributeKeykey$)
$\wedge$ BoundVariableInMethodBody($TextFigure, setAttribute, Objectvalue$)
$\wedge$ BoundVariableInMethodBody($SVGImage, setAttribute, AttributeKeykey$)
$\wedge$ BoundVariableInMethodBody($SVGImage, setAttribute, Objectvalue$)
$\wedge$ BoundVariableInMethodBody($SVGPath, setAttribute, AttributeKeykey$)
$\wedge$ BoundVariableInMethodBody($SVGPath, setAttribute, Objectvalue$)
$\wedge$ BoundVariableInMethodBody($BezierFigure, setAttribute, AttributeKeykey$)
$\wedge$ BoundVariableInMethodBody($BezierFigure, setAttribute, Objectvalue$)
$\wedge \neg$ExistsType($ContainsVisitor$)
$\wedge$ BoundVariableInMethodBody($GraphicalCompositeFigure, contains, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($EllipseFigure, contains, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($DiamondFigure, contains, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($RectangleFigure, contains, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($RoundRectangleFigure, contains, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($TriangleFigure, contains, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($TextAreaFigure, contains, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($TextFigure, contains, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($SVGImage, contains, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($AbstractCompositeFigure, contains, Point2D.Doublep$)
$\wedge$ BoundVariableInMethodBody($BezierFigure, contains, Point2D.Doublep$)
$\wedge$ ExistsType($AbstractFigure$)
$\wedge \neg$ExistsType($BasicTransformVisitor$)
$\wedge$ BoundVariableInMethodBody($AbstractCompositeFigure, basicTransform, AffineTransformtx$)
$\wedge$ BoundVariableInMethodBody($GraphicalCompositeFigure, basicTransform, AffineTransformtx$)
$\wedge$ BoundVariableInMethodBody($EllipseFigure, basicTransform, AffineTransformtx$)
$\wedge$ BoundVariableInMethodBody($DiamondFigure, basicTransform, AffineTransformtx$)
$\wedge$ BoundVariableInMethodBody($RectangleFigure, basicTransform, AffineTransformtx$)
$\wedge$ BoundVariableInMethodBody($RoundRectangleFigure, basicTransform, AffineTransformtx$)
$\wedge$ BoundVariableInMethodBody($TriangleFigure, basicTransform, AffineTransformtx$)
$\wedge$ BoundVariableInMethodBody($BezierFigure, basicTransform, AffineTransformtx$)
$\wedge$ BoundVariableInMethodBody($TextAreaFigure, basicTransform, AffineTransformtx$)
$\wedge$ BoundVariableInMethodBody($TextFigure, basicTransform, AffineTransformtx$)
$\wedge$ BoundVariableInMethodBody($SVGImage, basicTransform, AffineTransformtx$)
$\wedge$ BoundVariableInMethodBody($SVGPath, basicTransform, AffineTransformtx$)
$\wedge$ BoundVariableInMethodBody($LineConnectionFigure, basicTransform, AffineTransformtx$)
$\wedge$ ExistsMethodDefinitionWithParams($AbstractFigure, removeNotify, [Drawingd]$)
$\wedge$ ExistsAbstractMethod($AbstractFigure, removeNotify$)
$\wedge \neg$IsInheritedMethod($AbstractFigure, removeNotifyTmpVC$)
$\wedge \neg$IsInheritedMethodWithParams($AbstractFigure, removeNotifyTmpVC, [Drawingd]$)
$\wedge \neg$ExistsMethodDefinitionWithParams($AbstractFigure, removeNotifyTmpVC, [Drawingd]$)
$\wedge$ HasReturnType($AbstractFigure, removeNotify, Void$)
$\wedge$ ExistsMethodDefinition($AbstractFigure, removeNotify$)
$\wedge$ ExistsMethodDefinition($LabeledLineConnectionFigure, removeNotify$)
$\wedge$ ExistsMethodDefinition($GraphicalCompositeFigure, removeNotify$)
$\wedge$ ExistsMethodDefinition($EllipseFigure, removeNotify$)
$\wedge$ ExistsMethodDefinition($DiamondFigure, removeNotify$)
$\wedge$ ExistsMethodDefinition($RectangleFigure, removeNotify$)
$\wedge$ ExistsMethodDefinition($RoundRectangleFigure, removeNotify$)
$\wedge$ ExistsMethodDefinition($TriangleFigure, removeNotify$)
$\wedge$ ExistsMethodDefinition($TextFigure, removeNotify$)
$\wedge$ ExistsMethodDefinition($BezierFigure, removeNotify$)
$\wedge$ ExistsMethodDefinition($TextAreaFigure, removeNotify$)
$\wedge$ ExistsMethodDefinition($NodeFigure, removeNotify$)
$\wedge$ ExistsMethodDefinition($SVGImage, removeNotify$)
$\wedge$ ExistsMethodDefinition($DependencyFigure, removeNotify$)
$\wedge$ ExistsMethodDefinition($LineConnectionFigure, removeNotify$)

$\wedge$ ¬ExistsMethodDefinition($AbstractFigure, removeNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($LabeledLineConnectionFigure, removeNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($AbstractCompositeFigure, removeNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($GraphicalCompositeFigure, removeNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($EllipseFigure, removeNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($DiamondFigure, removeNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($RectangleFigure, removeNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($RoundRectangleFigure, removeNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($TriangleFigure, removeNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($TextFigure, removeNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($BezierFigure, removeNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($TextAreaFigure, removeNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($NodeFigure, removeNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($SVGImage, removeNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($SVGPath, removeNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($DependencyFigure, removeNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($LineConnectionFigure, removeNotifyTmpVC$)
$\wedge$ ExistsMethodDefinitionWithParams($AbstractFigure, addNotify, [Drawingd]$)
$\wedge$ ExistsAbstractMethod($AbstractFigure, addNotify$)
$\wedge$ ¬IsInheritedMethod($AbstractFigure, addNotifyTmpVC$)
$\wedge$ ¬IsInheritedMethodWithParams($AbstractFigure, addNotifyTmpVC, [Drawingd]$)
$\wedge$ ¬ExistsMethodDefinitionWithParams($AbstractFigure, addNotifyTmpVC, [Drawingd]$)
$\wedge$ HasReturnType($AbstractFigure, addNotify, Void$)
$\wedge$ ExistsMethodDefinition($AbstractFigure, addNotify$)
$\wedge$ ExistsMethodDefinition($LabeledLineConnectionFigure, addNotify$)
$\wedge$ ExistsMethodDefinition($GraphicalCompositeFigure, addNotify$)
$\wedge$ ExistsMethodDefinition($EllipseFigure, addNotify$)
$\wedge$ ExistsMethodDefinition($DiamondFigure, addNotify$)
$\wedge$ ExistsMethodDefinition($RectangleFigure, addNotify$)
$\wedge$ ExistsMethodDefinition($RoundRectangleFigure, addNotify$)
$\wedge$ ExistsMethodDefinition($TriangleFigure, addNotify$)
$\wedge$ ExistsMethodDefinition($BezierFigure, addNotify$)
$\wedge$ ExistsMethodDefinition($TextAreaFigure, addNotify$)
$\wedge$ ExistsMethodDefinition($SVGImage, addNotify$)
$\wedge$ ¬ExistsMethodDefinition($AbstractFigure, addNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($LabeledLineConnectionFigure, addNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($AbstractCompositeFigure, addNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($GraphicalCompositeFigure, addNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($EllipseFigure, addNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($DiamondFigure, addNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($RectangleFigure, addNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($RoundRectangleFigure, addNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($TriangleFigure, addNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($TextFigure, addNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($BezierFigure, addNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($TextAreaFigure, addNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($NodeFigure, addNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($SVGImage, addNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($SVGPath, addNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($DependencyFigure, addNotifyTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($LineConnectionFigure, addNotifyTmpVC$)
$\wedge$ ExistsMethodDefinitionWithParams($AbstractFigure, findFigureInside, [Point2D.Doublep]$)
$\wedge$ ExistsAbstractMethod($AbstractFigure, findFigureInside$)
$\wedge$ ¬IsInheritedMethod($AbstractFigure, findFigureInsideTmpVC$)
$\wedge$ ¬IsInheritedMethodWithParams($AbstractFigure, findFigureInsideTmpVC, [Point2D.Doublep]$)
$\wedge$ ¬ExistsMethodDefinitionWithParams($AbstractFigure, findFigureInsideTmpVC, [Point2D.Doublep]$)
$\wedge$ HasReturnType($AbstractFigure, findFigureInside, Figure$)
$\wedge$ ExistsMethodDefinition($AbstractFigure, findFigureInside$)
$\wedge$ ExistsMethodDefinition($EllipseFigure, findFigureInside$)
$\wedge$ ExistsMethodDefinition($DiamondFigure, findFigureInside$)
$\wedge$ ExistsMethodDefinition($RectangleFigure, findFigureInside$)
$\wedge$ ExistsMethodDefinition($RoundRectangleFigure, findFigureInside$)
$\wedge$ ExistsMethodDefinition($TriangleFigure, findFigureInside$)
$\wedge$ ExistsMethodDefinition($TextAreaFigure, findFigureInside$)
$\wedge$ ExistsMethodDefinition($SVGImage, findFigureInside$)
$\wedge$ ¬ExistsMethodDefinition($AbstractFigure, findFigureInsideTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($LabeledLineConnectionFigure, findFigureInsideTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($AbstractCompositeFigure, findFigureInsideTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($GraphicalCompositeFigure, findFigureInsideTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($EllipseFigure, findFigureInsideTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($DiamondFigure, findFigureInsideTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($RectangleFigure, findFigureInsideTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($RoundRectangleFigure, findFigureInsideTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($TriangleFigure, findFigureInsideTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($TextFigure, findFigureInsideTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($BezierFigure, findFigureInsideTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($TextAreaFigure, findFigureInsideTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($NodeFigure, findFigureInsideTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($SVGImage, findFigureInsideTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($SVGPath, findFigureInsideTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($DependencyFigure, findFigureInsideTmpVC$)
$\wedge$ ¬ExistsMethodDefinition($LineConnectionFigure, findFigureInsideTmpVC$)
$\wedge$ ExistsMethodDefinitionWithParams($AbstractFigure, setAttribute, [AttributeKeykey; Objectvalue]$)
$\wedge$ ExistsAbstractMethod($AbstractFigure, setAttribute$)
$\wedge$ ¬IsInheritedMethod($AbstractFigure, setAttributeTmpVC$)
$\wedge$ ¬IsInheritedMethodWithParams($AbstractFigure, setAttributeTmpVC, [AttributeKeykey; Objectvalue]$)
$\wedge$ ¬ExistsMethodDefinitionWithParams($AbstractFigure, setAttributeTmpVC, [AttributeKeykey; Objectvalue]$)
$\wedge$ HasReturnType($AbstractFigure, setAttribute, Void$)
$\wedge$ ExistsMethodDefinition($AbstractFigure, setAttribute$)
$\wedge$ ExistsMethodDefinition($AbstractCompositeFigure, setAttribute$)

$\wedge$ ExistsMethodDefinition($GraphicalCompositeFigure, setAttribute$)
$\wedge$ ExistsMethodDefinition($EllipseFigure, setAttribute$)
$\wedge$ ExistsMethodDefinition($DiamondFigure, setAttribute$)
$\wedge$ ExistsMethodDefinition($RectangleFigure, setAttribute$)
$\wedge$ ExistsMethodDefinition($RoundRectangleFigure, setAttribute$)
$\wedge$ ExistsMethodDefinition($TriangleFigure, setAttribute$)
$\wedge$ ExistsMethodDefinition($TextAreaFigure, setAttribute$)
$\wedge$ ExistsMethodDefinition($SVGImage, setAttribute$)
$\wedge$ ExistsMethodDefinition($SVGPath, setAttribute$)
$\wedge \neg$ExistsMethodDefinition($AbstractFigure, setAttributeTmpVC$)
$\wedge \neg$ExistsMethodDefinition($LabeledLineConnectionFigure, setAttributeTmpVC$)
$\wedge \neg$ExistsMethodDefinition($AbstractCompositeFigure, setAttributeTmpVC$)
$\wedge \neg$ExistsMethodDefinition($GraphicalCompositeFigure, setAttributeTmpVC$)
$\wedge \neg$ExistsMethodDefinition($EllipseFigure, setAttributeTmpVC$)
$\wedge \neg$ExistsMethodDefinition($DiamondFigure, setAttributeTmpVC$)
$\wedge \neg$ExistsMethodDefinition($RectangleFigure, setAttributeTmpVC$)
$\wedge \neg$ExistsMethodDefinition($RoundRectangleFigure, setAttributeTmpVC$)
$\wedge \neg$ExistsMethodDefinition($TriangleFigure, setAttributeTmpVC$)
$\wedge \neg$ExistsMethodDefinition($TextFigure, setAttributeTmpVC$)
$\wedge \neg$ExistsMethodDefinition($BezierFigure, setAttributeTmpVC$)
$\wedge \neg$ExistsMethodDefinition($TextAreaFigure, setAttributeTmpVC$)
$\wedge \neg$ExistsMethodDefinition($NodeFigure, setAttributeTmpVC$)
$\wedge \neg$ExistsMethodDefinition($SVGImage, setAttributeTmpVC$)
$\wedge \neg$ExistsMethodDefinition($SVGPath, setAttributeTmpVC$)
$\wedge \neg$ExistsMethodDefinition($DependencyFigure, setAttributeTmpVC$)
$\wedge \neg$ExistsMethodDefinition($LineConnectionFigure, setAttributeTmpVC$)
$\wedge$ ExistsMethodDefinitionWithParams($AbstractFigure, contains, [Point2D.Doublep]$)
$\wedge$ ExistsAbstractMethod($AbstractFigure, contains$)
$\wedge \neg$IsInheritedMethod($AbstractFigure, containsTmpVC$)
$\wedge \neg$IsInheritedMethodWithParams($AbstractFigure, containsTmpVC, [Point2D.Doublep]$)
$\wedge \neg$ExistsMethodDefinitionWithParams($AbstractFigure, containsTmpVC, [Point2D.Doublep]$)
$\wedge$ HasReturnType($AbstractFigure, contains, Boolean$)
$\wedge$ ExistsMethodDefinition($AbstractFigure, contains$)
$\wedge$ ExistsMethodDefinition($GraphicalCompositeFigure, contains$)
$\wedge$ ExistsMethodDefinition($EllipseFigure, contains$)
$\wedge$ ExistsMethodDefinition($DiamondFigure, contains$)
$\wedge$ ExistsMethodDefinition($RectangleFigure, contains$)
$\wedge$ ExistsMethodDefinition($RoundRectangleFigure, contains$)
$\wedge$ ExistsMethodDefinition($TriangleFigure, contains$)
$\wedge$ ExistsMethodDefinition($TextAreaFigure, contains$)
$\wedge$ ExistsMethodDefinition($SVGImage, contains$)
$\wedge \neg$ExistsMethodDefinition($AbstractFigure, containsTmpVC$)
$\wedge \neg$ExistsMethodDefinition($LabeledLineConnectionFigure, containsTmpVC$)
$\wedge \neg$ExistsMethodDefinition($AbstractCompositeFigure, containsTmpVC$)
$\wedge \neg$ExistsMethodDefinition($GraphicalCompositeFigure, containsTmpVC$)
$\wedge \neg$ExistsMethodDefinition($EllipseFigure, containsTmpVC$)
$\wedge \neg$ExistsMethodDefinition($DiamondFigure, containsTmpVC$)
$\wedge \neg$ExistsMethodDefinition($RectangleFigure, containsTmpVC$)
$\wedge \neg$ExistsMethodDefinition($RoundRectangleFigure, containsTmpVC$)
$\wedge \neg$ExistsMethodDefinition($TriangleFigure, containsTmpVC$)
$\wedge \neg$ExistsMethodDefinition($TextFigure, containsTmpVC$)
$\wedge \neg$ExistsMethodDefinition($BezierFigure, containsTmpVC$)
$\wedge \neg$ExistsMethodDefinition($TextAreaFigure, containsTmpVC$)
$\wedge \neg$ExistsMethodDefinition($NodeFigure, containsTmpVC$)
$\wedge \neg$ExistsMethodDefinition($SVGImage, containsTmpVC$)
$\wedge \neg$ExistsMethodDefinition($SVGPath, containsTmpVC$)
$\wedge \neg$ExistsMethodDefinition($DependencyFigure, containsTmpVC$)
$\wedge \neg$ExistsMethodDefinition($LineConnectionFigure, containsTmpVC$)
$\wedge$ ExistsClass($AbstractFigure$)
$\wedge$ IsAbstractClass($AbstractFigure$)
$\wedge$ ExistsMethodDefinitionWithParams($AbstractFigure, basicTransform, [AffineTransformtx]$)
$\wedge$ ExistsAbstractMethod($AbstractFigure, basicTransform$)
$\wedge \neg$IsInheritedMethod($AbstractFigure, basicTransformTmpVC$)
$\wedge \neg$IsInheritedMethodWithParams($AbstractFigure, basicTransformTmpVC, [AffineTransformtx]$)
$\wedge \neg$ExistsMethodDefinitionWithParams($AbstractFigure, basicTransformTmpVC, [AffineTransformtx]$)
$\wedge$AllSubclasses($AbstractFigure, [LabeledLineConnectionFigure; AbstractCompositeFigure; GraphicalCompositeFigure; EllipseFigure; DiamondFigure$
$\wedge$ HasReturnType($AbstractFigure, basicTransform, Void$)
$\wedge \neg$IsPrivate($AbstractFigure, basicTransform$)
$\wedge \neg$IsPrivate($LabeledLineConnectionFigure, basicTransform$)
$\wedge \neg$IsPrivate($AbstractCompositeFigure, basicTransform$)
$\wedge \neg$IsPrivate($GraphicalCompositeFigure, basicTransform$)
$\wedge \neg$IsPrivate($EllipseFigure, basicTransform$)
$\wedge \neg$IsPrivate($DiamondFigure, basicTransform$)
$\wedge \neg$IsPrivate($RectangleFigure, basicTransform$)
$\wedge \neg$IsPrivate($RoundRectangleFigure, basicTransform$)
$\wedge \neg$IsPrivate($TriangleFigure, basicTransform$)
$\wedge \neg$IsPrivate($TextFigure, basicTransform$)
$\wedge \neg$IsPrivate($BezierFigure, basicTransform$)
$\wedge \neg$IsPrivate($TextAreaFigure, basicTransform$)
$\wedge \neg$IsPrivate($NodeFigure, basicTransform$)
$\wedge \neg$IsPrivate($SVGImage, basicTransform$)
$\wedge \neg$IsPrivate($SVGPath, basicTransform$)
$\wedge \neg$IsPrivate($DependencyFigure, basicTransform$)
$\wedge \neg$IsPrivate($LineConnectionFigure, basicTransform$)
$\wedge$ ExistsMethodDefinition($AbstractFigure, basicTransform$)
$\wedge$ ExistsMethodDefinition($AbstractCompositeFigure, basicTransform$)
$\wedge$ ExistsMethodDefinition($GraphicalCompositeFigure, basicTransform$)
$\wedge$ ExistsMethodDefinition($EllipseFigure, basicTransform$)
$\wedge$ ExistsMethodDefinition($DiamondFigure, basicTransform$)
$\wedge$ ExistsMethodDefinition($RectangleFigure, basicTransform$)

$\wedge$ ExistsMethodDefinition($RoundRectangleFigure, basicTransform$)
$\wedge$ ExistsMethodDefinition($TriangleFigure, basicTransform$)
$\wedge$ ExistsMethodDefinition($TextAreaFigure, basicTransform$)
$\wedge$ ExistsMethodDefinition($SVGImage, basicTransform$)
$\wedge$ ExistsMethodDefinition($SVGPath, basicTransform$)
$\wedge \neg$ExistsMethodDefinition($AbstractFigure, basicTransformTmpVC$)
$\wedge \neg$ExistsMethodDefinition($LabeledLineConnectionFigure, basicTransformTmpVC$)
$\wedge \neg$ExistsMethodDefinition($AbstractCompositeFigure, basicTransformTmpVC$)
$\wedge \neg$ExistsMethodDefinition($GraphicalCompositeFigure, basicTransformTmpVC$)
$\wedge \neg$ExistsMethodDefinition($EllipseFigure, basicTransformTmpVC$)
$\wedge \neg$ExistsMethodDefinition($DiamondFigure, basicTransformTmpVC$)
$\wedge \neg$ExistsMethodDefinition($RectangleFigure, basicTransformTmpVC$)
$\wedge \neg$ExistsMethodDefinition($RoundRectangleFigure, basicTransformTmpVC$)
$\wedge \neg$ExistsMethodDefinition($TriangleFigure, basicTransformTmpVC$)
$\wedge \neg$ExistsMethodDefinition($TextFigure, basicTransformTmpVC$)
$\wedge \neg$ExistsMethodDefinition($BezierFigure, basicTransformTmpVC$)
$\wedge \neg$ExistsMethodDefinition($TextAreaFigure, basicTransformTmpVC$)
$\wedge \neg$ExistsMethodDefinition($NodeFigure, basicTransformTmpVC$)
$\wedge \neg$ExistsMethodDefinition($SVGImage, basicTransformTmpVC$)
$\wedge \neg$ExistsMethodDefinition($SVGPath, basicTransformTmpVC$)
$\wedge \neg$ExistsMethodDefinition($DependencyFigure, basicTransformTmpVC$)
$\wedge \neg$ExistsMethodDefinition($LineConnectionFigure, basicTransformTmpVC$)
$\wedge$ ExistsType($GraphicalCompositeFigure$)
$\wedge$ ExistsClass($GraphicalCompositeFigure$)
$\wedge$ IsSubType($GraphicalCompositeFigure, AbstractCompositeFigure$)
$\wedge \neg$ExistsMethodDefinition($GraphicalCompositeFigure, findFigureInside$)
$\wedge$ ExistsType($NodeFigure$)
$\wedge$ ExistsClass($NodeFigure$)
$\wedge$ IsSubType($NodeFigure, TextFigure$)
$\wedge \neg$ExistsMethodDefinition($NodeFigure, addNotify$)
$\wedge \neg$ExistsMethodDefinition($NodeFigure, basicTransform$)
$\wedge \neg$ExistsMethodDefinition($NodeFigure, setAttribute$)
$\wedge \neg$ExistsMethodDefinition($NodeFigure, findFigureInside$)
$\wedge \neg$ExistsMethodDefinition($NodeFigure, contains$)
$\wedge$ ExistsMethodDefinition($TextFigure, addNotify$)
$\wedge$ ExistsMethodDefinition($TextFigure, basicTransform$)
$\wedge$ ExistsMethodDefinition($TextFigure, setAttribute$)
$\wedge$ ExistsMethodDefinition($TextFigure, findFigureInside$)
$\wedge$ ExistsMethodDefinition($TextFigure, contains$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextFigure, addNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextFigure, basicTransform, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextFigure, setAttribute, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextFigure, findFigureInside, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($TextFigure, contains, this$)
$\wedge$ ExistsType($DependencyFigure$)
$\wedge$ ExistsClass($DependencyFigure$)
$\wedge$ IsSubType($DependencyFigure, LineConnectionFigure$)
$\wedge \neg$ExistsMethodDefinition($DependencyFigure, addNotify$)
$\wedge \neg$ExistsMethodDefinition($DependencyFigure, basicTransform$)
$\wedge \neg$ExistsMethodDefinition($DependencyFigure, setAttribute$)
$\wedge \neg$ExistsMethodDefinition($DependencyFigure, findFigureInside$)
$\wedge \neg$ExistsMethodDefinition($DependencyFigure, contains$)
$\wedge$ ExistsMethodDefinition($LineConnectionFigure, addNotify$)
$\wedge$ ExistsMethodDefinition($LineConnectionFigure, basicTransform$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($LineConnectionFigure, addNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($LineConnectionFigure, basicTransform, this$)
$\wedge$ ExistsType($LabeledLineConnectionFigure$)
$\wedge$ ExistsClass($LabeledLineConnectionFigure$)
$\wedge$ IsSubType($LabeledLineConnectionFigure, BezierFigure$)
$\wedge \neg$ExistsMethodDefinition($LabeledLineConnectionFigure, basicTransform$)
$\wedge \neg$ExistsMethodDefinition($LabeledLineConnectionFigure, setAttribute$)
$\wedge \neg$ExistsMethodDefinition($LabeledLineConnectionFigure, findFigureInside$)
$\wedge \neg$ExistsMethodDefinition($LabeledLineConnectionFigure, contains$)
$\wedge$ ExistsMethodDefinition($BezierFigure, basicTransform$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($BezierFigure, basicTransform, this$)
$\wedge$ ExistsType($SVGPath$)
$\wedge$ ExistsClass($SVGPath$)
$\wedge$ IsSubType($SVGPath, AbstractCompositeFigure$)
$\wedge \neg$ExistsMethodDefinition($SVGPath, addNotify$)
$\wedge \neg$ExistsMethodDefinition($SVGPath, removeNotify$)
$\wedge \neg$ExistsMethodDefinition($SVGPath, findFigureInside$)
$\wedge \neg$ExistsMethodDefinition($SVGPath, contains$)
$\wedge$ ExistsMethodDefinition($AbstractCompositeFigure, addNotify$)
$\wedge$ ExistsMethodDefinition($AbstractCompositeFigure, removeNotify$)
$\wedge$ ExistsMethodDefinition($AbstractCompositeFigure, findFigureInside$)
$\wedge$ ExistsMethodDefinition($AbstractCompositeFigure, contains$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($AbstractCompositeFigure, addNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($AbstractCompositeFigure, removeNotify, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($AbstractCompositeFigure, findFigureInside, this$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($AbstractCompositeFigure, contains, this$)
$\wedge$ ExistsType($LineConnectionFigure$)
$\wedge$ ExistsClass($LineConnectionFigure$)
$\wedge$ IsSubType($LineConnectionFigure, BezierFigure$)
$\wedge \neg$ExistsMethodDefinition($LineConnectionFigure, findFigureInside$)
$\wedge \neg$ExistsMethodDefinition($LineConnectionFigure, setAttribute$)
$\wedge \neg$ExistsMethodDefinition($LineConnectionFigure, contains$)
$\wedge$ ExistsMethodDefinition($BezierFigure, findFigureInside$)
$\wedge$ ExistsMethodDefinition($BezierFigure, setAttribute$)
$\wedge$ ExistsMethodDefinition($BezierFigure, contains$)
$\wedge$ AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded($BezierFigure, findFigureInside, this$)

$\wedge$ `AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded`$(BezierFigure, setAttribute, this)$
$\wedge$ `AllInvokedMethodsWithParameterOInBodyOfMAreNotOverloaded`$(BezierFigure, contains, this)$

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

MINES Nantes
Institut Mines-Télécom

Thèse de Doctorat

Akram AJOULI

Vues et Transformations de Programmes pour la Modularité des Évolutions

Views and Program transformations for modular maintenances

Résumé

La maintenance consomme une grande partie du coût de développement des logiciels ce qui rend l'optimisation de ce coût parmi les enjeux importants dans le monde du génie logiciel. Dans cette thèse nous visons à optimiser ce coût par rendre ces maintenances modulaires. Pour atteindre cet objectif, nous définissons des transformations des architectures des programmes qui permettent de transformer le programme à maintenir vers une architecture qui facilite la tâche de maintenance voulue. Nous nous concentrons plus sur la transformation entre les architectures à propriétés de modularité duales tels que les patrons de conception Composite et Visiteur. Dans ce contexte, nous définissons une transformation automatique et réversible basée sur le refactoring entre un programme structuré selon le Composite et sa structure Visiteur correspondante. Cette transformation est validée par la génération d'une précondition qui garantit statiquement sa réussite. Elle est aussi adaptée afin qu'elle prenne en compte la transformation de quatre variations du patron Composite et est validée sur le programme JHotDraw qui comporte ces quatre variations. Nous définissons aussi une transformation réversible au sein du patron Singleton afin de pouvoir bénéficier de l'optimisation par l'introduction de ce patron et la souplesse par sa suppression selon les exigences de l'utilisateur du logiciel.

Mots clés

maintenance modulaires, patrons de conception, refactoring,transformation des programmes, préconditions minimales.

Abstract

Maintenance consumes a large part of the cost of software development which makes the optimization of that cost among the important issues in the world of software engineering. In this thesis we aim to optimize this cost by making these maintenances modular. To achieve this goal, we define transformations of program architectures that allow to transform a program to maintain into an architecture that facilitates the maintenance tasks required. We focus on transformation between architectures having dual modularity properties such as Composite and Visitor design patterns. In this context, we define an automatic and reversible transformation based on refactoring between a program structured according to the Composite structure and its corresponding Visitor structure. This transformation is validated by generating a precondition which guarantees statically its success. It is also adapted to take into account the transformation of four variations of Composite pattern and it is then applied to JHotDraw program in which these four variations occur. We define also a reversible transformation in the Singleton pattern to benefit from optimization by introducing this pattern and flexibility by its suppression according to the requirements of the software user.

Key Words

modular maintenance, design patterns, refactoring, program transformation, minimal precondition.